NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

JOÃO MIGUEL POMAR MONTEIRO

Degree in Computer Science and Engineering

A MULTI LEVEL DHT APPROACH THROUGH HIERARCHICAL NAMING

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

DEPARTMENT OF
COMPUTER SCIENCE

A MULTI LEVEL DHT APPROACH THROUGH HIERARCHICAL NAMING

JOÃO MIGUEL POMAR MONTEIRO

Degree in Computer Science and Engineering

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

Examination Committee:

Chair:	Artur Miguel de Andrade Vieira Dias *Assistant Professor, NOVA University Lisbon*
Rapporteur:	Ricardo Vilaça *Senior Researcher, University of Minho*
Adviser:	João Carlos Antunes Leitão *Assistant Professor, NOVA University Lisbon*

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

A Multi Level DHT approach through hierarchical naming

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

To family and friends.

Acknowledgements

The work presented here would not have been possible without the support of several people with whom I worked during the course of this thesis. Firstly, I would like to thank my advisor João Leitão, for his insightful guidance that helped me in pursuing the objectives of this work.

I want to thank Pedro Ákos Costa, a doctoral student here at NOVA that without his continuous support and motivation, none of this would have been possible. Pedro contributed with additional ideas while developing the work presented here and some of the basis for the experimental tools used.

Moreover, I would like to thank the Department of Computer Science of NOVA School of Science & Technology and NOVA LINCS that provided tools to develop the work pursued here. Additionally, I want to send a special thanks to my colleagues at the research center, as they create a great work environment as they are always keen to help and unwind when needed.

I want to deeply thank my family for providing all the support they can, not only during this work but also during the whole course. Finally, I want to thank my friends that helped me in maintaining my motivation even in the most distressing times.

"If everything seems under control, you're just not going fast enough." (Mario Andretti)

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 the 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.

In this work, we developed two different schemes to organize a network topology, using the Kademlia DHT as the underlying overlay and the case study for this work. In these schemes, nodes organize themselves through the use of deterministic labels that are added to their identifiers, which encode a property of the node related with an abstract notion of distance (e.g., geographical, similarities between the applications being executed, etc.). With this topology change, it is possible to find nearby content faster with a lower hop count, maintaining an acceptable hop count to distant content. The first scheme, Soft Partition, prepends the label to nodes' identifiers, creating a bias in the identifier space with nodes sharing the same label becoming closer in this space. The second scheme, Hard Partition, builds a DHT for each label in the network, creating a crisp division between nodes with different labels, which are assisted by an external service to enable communication between the different DHTs. 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 necessários 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.

Neste trabalho, desenvolvemos dois esquemas de organização topológica usando a rede estruturada Kademlia como rede sobreposta subjacente e caso de estudo para este trabalho. Nestes esquemas, os participantes organizam-se através do uso de rótulos determinísticos adicionados aos identificadores que codificam uma propriedade do nó relacionada com uma noção abstrata de distância (e.g., localização geográfica, semelhança entre aplicações a ser executadas, etc.). Com base nesta organização, é possível encontrar eficientemente recursos próximos com um baixo número de saltos, mantendo em simultâneo um número aceitável de saltos necessários para encontrar conteúdos longínquos. O primeiro dos esquemas, particionamento suave, adiciona efetivamente o rótulo ao identificador criando assim uma alteração do espaço de identificadores, onde participantes com o mesmo rótulo ficam mais próximos no espaço. O segundo esquema, particionamento rígido, forma uma DHT para cada rótulo na rede criando assim uma divisão entre participantes que recorrem a um serviço externo para comunicar entre si. 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

Annexes

List of Figures

List of Tables

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 [21]. 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 [37].

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), each participant 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, which 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.

In unstructured overlays [22, 24, 43], 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 [3, 20], where they exchange messages randomly between themselves to propagate information or locate resources on the network.

Structured overlays [27, 28], unlike their unstructured counterpart, 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 (DHTs) [14, 30, 39, 40, 46] 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 [4], and more recently, the InterPlanetary File System [1]. The latter will serve as a 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 constraints, 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 peers 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 systems could benefit from the use of both structured and unstructured overlays to provide more stability in dynamic scenarios. While there is some literature exploring the use of these overlays [29, 34], it remains an open question on how to better take advantage of such an approach. Multi-Level DHTs [10] are another case of combining multiple overlays where each of the levels imposes different constraints (e.g., each level is a separate DHT that corresponds to some latency requirements), which creates low diameter DHTs in the lower levels that produce more efficient queries if the access to higher levels is not needed. However, this type of approach can add extra management complexity, especially in highly dynamic networks, as nodes need to find the better suited DHT to join.

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 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 systems in realistic settings.

1.3 Contributions

The main contributions presented in this work are the following:

- Two distinct DHT topology organization schemes that leverage an abstract notion of locality in order to create more efficient paths in the network, thus accelerating search queries. The Soft Partition scheme applies prefixes to node and content identifiers that bias the topology towards the dimension of locality desirable for the system. The Hard Partition scheme builds a DHT for each label in the network, creating smaller diameter networks. Indexers scattered throughout the network serve as bridges between DHTs, allowing communication between them.
- A concrete implementation of the aforementioned topology organization schemes over the Kademlia DHT [30].
- An experimental comparison between our two schemes and the baseline Kademlia DHT, using an emulated network under multiple scenarios, which shows the benefits of biasing the topology when there is an access pattern directed towards locality.

1.3.1 Research Context

The work conducted in the context of this thesis was 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 [33] targeted at the research of novel Multi-Level DHT designs and evaluation.

1.3.2 Publications

The work presented here produced the following publication:

- P-KAD: Enriquecer o Kademlia com Particionamento. (Communication)

  João Monteiro, Pedro Ákos Costa, Alfonso de la Rocha, Yiannis Psaras and João Leitão.

  Comunications of the 12th Simpósio de Informática (INForum 2021), Lisbon, Portugal, Setember 2021

1.4 Document organization

The document is organized as follows:

- In Chapter 2 we discuss related work. We explain in more detail the generic architectural stack of P2P systems, different challenges that may arise, and study in more detail the differences between the structured and unstructured overlays. We provide an overview on different implementations of P2P networks, both structured and unstructured. Finally, we study in more detail a distributed file system architecture and the protocols used in its construction.

- In Chapter 3 we describe the developed solutions and detail the current implementations using the Kademlia DHT as the baseline.

- In Chapter 4 we present the experimental evaluation performed using an emulation platform.

- In Chapter 5 we present the conclusions and possible future work.

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 overlays (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 case study of this thesis (Section 2.5).

2.1 Peer-to-Peer Systems' Concepts

In a centralized architecture, a single computer or server holds all 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. [21].

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 better scalability compared to a centralized alternative, the coordinator can still become a major point of failure and a bottleneck as all operations 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 is that the amount of resources increases every time a node joins the network without the need for an infrastructure upgrade, due to their organic growth and no need for dedicated infrastructure [37].

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.

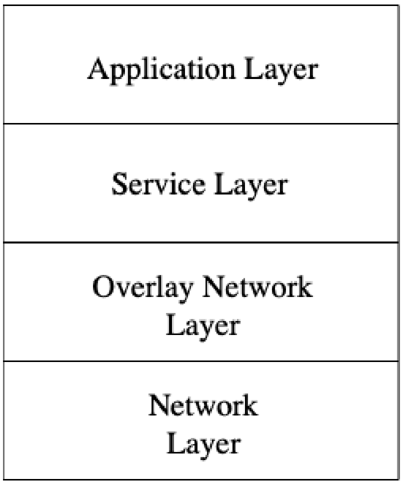


Figure 2.1: Peer-to-Peer generic architecture (adapted from [21])

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 [22, 30, 40]. 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 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 members of the system. As the system grows, this strategy becomes unfeasible due to 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 [8, 20] leverages the built overlays to execute services such as lookup protocols or message broadcasting, providing a higher level of abstraction for the layers above to leverage the underlying overlay.

Application Layer The Application layer exposes the end interface to the user and implements the application logic using the service layers below. IPFS [1], and BitTorrent[4] are examples of P2P applications.

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 participants had shared in the network, and directly download it from one or more peers. Rapidly, many other applications appeared that offered similar but improved services like Gnutella and Kazza [37]. Currently, one of the most famous P2P file sharing application is BitTorrent [4]. Using *trackers*, users 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.

Further ahead on 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 [23].

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 and generic 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 [3, 20]. In a gossip protocol, nodes collaborate to broadcast a message through the network, similarly to how rumors spread in a population. In order to disseminate messages, a node 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 [21]. These include, but are not limited to the following:

Eager-Push When 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 it learns of a message that has not received yet, it 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. Additionally, 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 that matches with the query and return an answer to the participant that issued it [21].

Exact match queries use unique identifiers from resources to locate them in the network. In order to issue this type of queries, a client 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.

Another 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 [27, 28].

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 order for a participant to locate a resource in these types of overlays, 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 overlays 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 [22], Overnesia [24], Cyclon [43], and Scamp [11].

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 [40], a P2P system that organizes peers in a ring topology, Kademlia [30] that organizes nodes in a binary tree, and Pastry [39] that creates a structured mesh.

In [16], 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 [32] 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 [29].

2.1.5 Main Challenges

In [35] the authors detail some relevant issues 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 where 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 [35], 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 (or in a more generic way, a resource) 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. However, their effectiveness 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 [41], 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 number of nodes leaves concurrently, the membership protocol may not be able to keep up and nodes may become unreachable in some cases due to all their neighbors disconnecting [31].

Unstructured overlays, due to their flexibility in managing the topology, are less susceptible to churn scenarios than 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, since the number of hops to locate a resource increases with the network size.

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

P2P systems can take into account the underlying topology to bias the neighbor sets [25]. 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. An issue that can be 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.

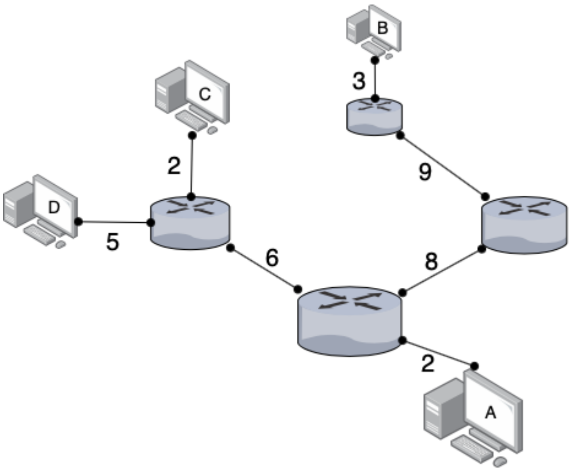
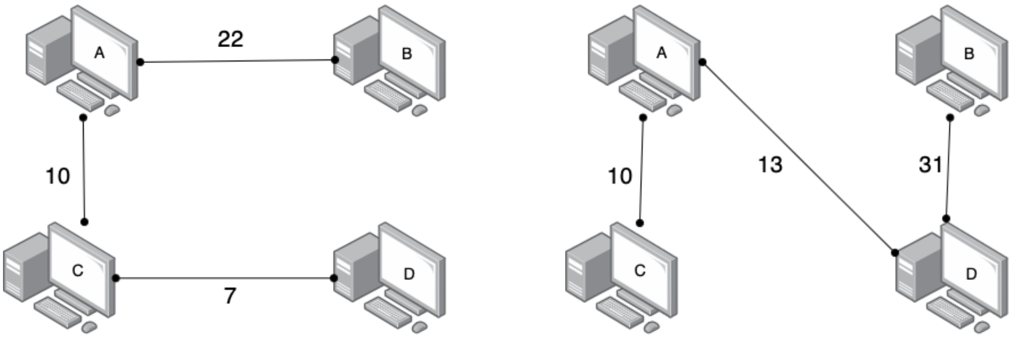


Figure 2.2: Network Topology



(a) Overlay paths are well optimized

(b) Topology Mismatch in overlay

Figure 2.3: Two possible overlay configurations.

2.1.5.4 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 [6].

Super-peer networks are a class of overlay that leverages the use of more stable and powerful nodes to create more efficient routing protocols [2]. 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.

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. Even though the main focus of our work is on structured overlays, it is also useful to study this type of overlays as they provide valuable lessons and techniques that can also be applied to the construction of more fault-tolerant structured overlays.

In these networks, 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 [43].

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 [22] uses a reactive strategy to update a part of each node's neighbors (effectively those that are used to exchange messages).

2.2.1 Examples from the Literature

2.2.1.1 Cyclon

Cyclon [43] uses a cyclic strategy to maintain the set encoding a node's neighbors (i.e., the partial view). Each entry in the 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 *n1*.

Node *n1* 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 it 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 receives the set of samples from *n1*.

A joining node *n2* 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 *n2* and decrease the TTL in each hop. Finally, *n2* 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, be too large to timely deal with highly unstable networks (e.g., churn conditions).

2.2.1.2 HyParView

HyParView [22] 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 configuration parameters associated. The *Active Random Walk Length* (ARWL) is the parameter that specifies the message's maximum TTL and the *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 *p*'s 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 the current size of the active view. The message has high priority if *p*'s active view is empty, otherwise is a 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 random node from its view. It only accepts a low priority request 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 [43]. 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 size, 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 [22] 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 fewer rounds to reestablish its reliability in highly unstable scenarios.

2.2.1.3 Overnesia

Overnesia [24] 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 that is responsible for maintaining an index of all regular nodes connected to them. 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 that act as replicas of each other. Each of these replica nodes possess 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 the 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 with all the cluster's members.

Nodes maintain a constant size neighboring set containing members of external clusters. To acquire these neighbors, a participant uses a random walk to find new nodes that also have empty space in their external neighbor set. If the random walk terminates without 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. Upon receiving this message, a node will wait a predefined amount of time and if no other division message was sent by another 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 contain 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 [24], Overnesia is able to handle high amounts of churn compared to other unstructured overlays and the Chord DHT [40], which will be discussed further ahead.

2.2.2 Discussion

Unstructured overlays 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 [25].

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, which allows lookup mechanisms to make efficient decisions 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 of the participants' routing tables. To each participant, a uniformly random unique identifier is assigned that defines the position of the node and allows its location in the network. This is usually done by using a consistent hashing function [5].

Structured Overlays are usually used as DHTs (Distributed Hash Tables)[40, 30] where resources are also assigned unique identifiers similarly to nodes. Therefore, each participant 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 in comparison with the overall network, and when a node enters or leaves the system, only some nodes are affected.

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 others 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 [35].

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 physical 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 [40] 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 traverse 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, for 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 that 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 that 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 [9], which improves load balancing and the possibility to execute range-queries.

Kademlia Kademlia [30] 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, which 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 it 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 the 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 one 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 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. It should be noted that when compared to classical DHTs (e.g, Chord [40] or Tapestry [47]) Kademlia nodes maintain information, on average, about a significantly higher number of nodes in their buckets.

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, it returns the stored value. 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 query 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, except 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 such, these searches are likely to hit cached entries first alleviating hot spots. It is expected that a lookup request will take at most $O(\log n)$ steps. Cached entries need to have an expiration time in order to avoid accessing outdated data.

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, in order to populate them and to be inserted into other nodes' buckets to become 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 it periodically, to compensate for nodes leaving and new nodes arriving with identifiers closer to the value's key.

Kademlia, compared with Chord [40], 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 to denial of service attacks. Kademlia implementations appear in file sharing networks such as BitTorrent clients [4] 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. However, in these schemes it 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 [14] 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 that 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 other 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 slots reserved to new entries 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 that 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 to transfer the resource to it. Therefore, insertions 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 operate 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, it 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 an 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 [47] 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 i^{th} level of the map and find the entry that corresponds to the digit in that position. This entry's node is the closest neighbor known, which ends with the searched prefix. When the entry to be used is empty, surrogate routing is used to choose the next best entry option. The routing protocol guarantees that every node can be found in $O(\log_b n)$ steps, being b the identifier size, and n the number of members in the DHT. 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 alternative overlay paths. 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 has been reached, where the only available next hop is the node itself. During this process, at each hop, 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 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 contains the older pointer, it will replace the entry and continue until the root is reached. Concurrently, 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 [10] organizes nodes in a hierarchical structure of clusters, identified by a unique identifier, which increase in diameter, and 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 (CDNs) with multiple writers and readers of the same content. In the original implementation described in the paper, Coral is built as a layer on top of Chord [40] lookup service, although authors argue that other DHTs can be used.

This system implements a *distributed sloppy hash table* (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 key 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 in the closest node to the resource's key.

A lookup searching for a specific key will traverse 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 counterparts 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 clusters that meet the round trip requirements are 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, in 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 be 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 topology restrictions imposed by the overlay design. Due to this intermediate step, in high churn scenarios, the network could become partitioned, and the routing infrastructure can therefore fail. In [31], 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 [47], 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 becomes 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 a node 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 grows, 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.

There is additional literature on enriching classical DHTs with locality-aware properties. In [13], the authors apply localization techniques such as GPS or IP locator services to Kademlia DHT in order to position nodes in a sphere, and by using distance measurements nodes prioritize participants that have the less distance to themselves. In [45], the authors use the node's Autonomous System Number (ASN) to build a prefix to the identifier, thus biasing node's position in the identifier space, which can be applied to various DHTs. While it might be easy to obtain a process ISN, it is unclear if ASNs are the adequate granularity of location for most applications. The authors in [18] build a super-peer network with each of them being responsible for the communication between the different geographical areas. All of these works explore a similar path to our work, which will be detailed in Chapter 3, but while these are strongly tied to geographical properties, our solutions can be generalized to be used in more abstract proximity metrics such as the application that the node is running.

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 [26].

2.4.1 Example for the Literature

2.4.1.1 Rollerchain

Rollerchain [34] is a DHT that uses a Chord [40] 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 [24] 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 increasing 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 it 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 dynamic systems, multiple nodes may join or leave, creating an unbalance in link distribution. To re-balance links, the cluster representative (i.e., the node with the smallest identifier within that cluster) is able to activate a procedure that redistributes connections.

In the experimental evaluation conducted in the original paper [34], the overlay management is higher than compared to Chord's. This is expected as Rollerchain has to maintain two types of overlays. When comparing load balancing results in scenarios where keys are distributed uniformly and others 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 [29], the authors experiment with building a P2P overlay using Pastry DHT [39] 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 any 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 [28]. 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 P2P distributed file system designed by Protocol Labs that seeks to connect all computing devices with the same system of files [1]. IPFS aims to create 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 [30] based implementation, which we will detail in Section 2.5.1.1.

We are now going to present and discuss the main ideas behind IPFS's architecture focusing on the routing and publishing protocols as detailed in the original paper [1], in [15], and the project's open source code[12].

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 [19]. 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 such, 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 [36]. 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 files: a Blob, which represents a single piece of data; a List, which can contain multiple blobs or other lists that represents a large file that needed to be divided into multiple blocks; a Tree, which can contain references to all other data types and represents a directory; Commits, which contain versions of objects. An example of an object is illustrated in Figure 2.4, with the C, T, L and B representing a Commit, Tree, List and Blob respectively.

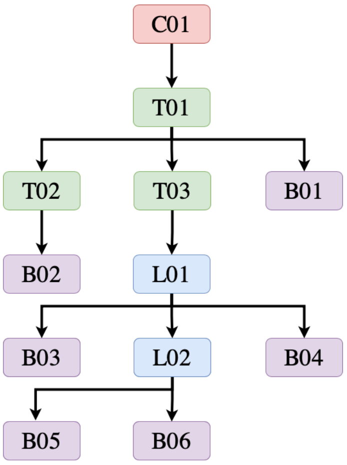


Figure 2.4: Object architecture example in IPFS

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 the task of file sharing difficult. 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 wants to access content published by another user, will only have to access the

IPNS link to redirect it to the more current data.

2.5.1.1 DHT protocol

The overlay structure of IPFS is built around a Kademlia based implementation using TCP. Some participants in IPFS might not join the DHT due to preference or due to connectivity issues (e.g., behind a NAT), only retrieving content by directly contacting other members. IPFS routing protocol maintains most of the characteristics detailed in the original paper [30] 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 for a node not to store mostly empty buckets, as nodes with very similar identifiers (i.e., with the 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, only evicting 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 the network (i.e., the key being the file's hash and the value being the participant's contact information).

In [15] the authors study and crawl the IPFS's Kademlia DHT to estimate metrics such as number of nodes concurrently in the system, number of connections each node has, node uptime, geographical location, etc. They conclude that the majority of nodes are deployed in short sessions as most of them are hosted in private machines (e.g., home computers and laptops), which is not ideal considering the low tolerance to churn that structured overlays have. One other side effect of having nodes deployed in private machines is that a large percentage of them are behind a NAT, which can make the nodes unreachable. From the crawls performed, the estimate for the concurrent number of nodes in the DHT is around 45 000 nodes, although the authors could only connect to roughly 6.5% of the observed nodes, which is a surprisingly low number.

The IPFS system handles this amount of churn and lack of connectivity by making every node maintain a sizable number of connections. From the study of the source code and configuration files present in [12], each DHT participant has a K of 20 nodes (i.e., 20 nodes per bucket in the routing table) with a maximum of 900 active connections concurrently, which for the current average system size can be excessive, as we will later see in Section 4.

Another interesting result from the crawler study, is the geographical distribution of nodes, which showed that most nodes on IPFS were based on China, Germany and the USA, although China had a low number of reachable nodes, with the USA having the highest number followed by France and Germany.

2.5.1.2 Bitswap protocol

Working in conjunction with the DHT, IPFS uses a protocol named Bitswap [36], based on BitTorrent's algorithm [4], to transfer and search data among peers in an unstructured fashion. When a node joins the network, starts a new session of this protocol and creates a *want list* of root identifiers for files that it is searching for. Bitswap architecture is represented in Figure 2.5. The ledger is used to check if the node can serve a request to nodes 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.

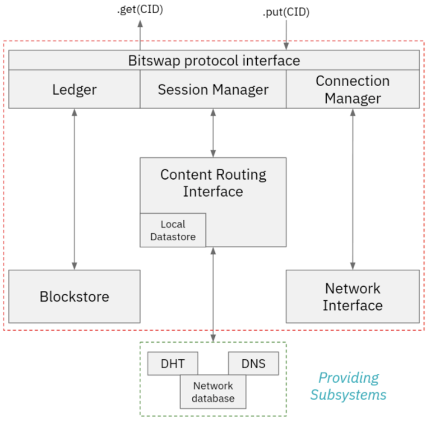


Figure 2.5: Bitswap architecture (adapted from [36])

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 an *IDONT-HAVE* otherwise. Upon receiving a *HAVE*, the initiator will only forward subsequent requests to these nodes, as the probability of having the rest of the 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 receives 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 [38]. 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 explored the use of an alternative DHT topology as a method for solving the aforementioned problem. The DHT design tries 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. Afterwards, 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 saw 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 standard DHT design may face scalability issues with its continuous growth, which motivated us to create an alternative design that could improve the performance of these types of use cases for P2P system.

In the following chapter we present our design proposals while also detailing and discussing their current implementations.

3

Multi Level DHT

In this chapter we present the design of our two proposals, capable of creating more efficient DHT routing paths to content, by biasing the DHT topology using a generic concept of locality (e.g., geographical region, running application, etc.).

We begin by introducing the system model assumptions, followed by the explanation and discussion of the two designs: Soft Partition and Hard Partition. We follow by providing details on the implementation of both solutions applied to the Kademlia DHT [30].

3.1 System Model and Assumptions

We now detail the minimal set of system assumptions we use regarding the design of our solutions.

1. In order for our two proposed solutions to bias the DHT topology, we resort to node labels that encode the defined concept of locality. In this work, we assume that every node can accurately choose its appropriate label by possessing and using a list containing all the possible label options.

2. A related system's assumption is that participants can obtain content identifiers with an out-of-band process. This assumption is based on the IPFS's operation [1].

3. To be able to join the system's network, every node needs to use some sort of bootstrap node. In our system model every node joins the network using the same set of bootstrap nodes, which do not fail to respond.

4. In real environments, nodes experience different latencies to every other node. Our system model assumes that, with a high probability, nodes that are closer in the underlying network topology have better latency between themselves compared with nodes further away.

5. It is assumed that all nodes that want to participate in the network are reachable by every other node, so that they can receive and reply to request messages.

A strong assumption that we have is that most of the accesses to content have a high correlation with the abstract notion of locality used. For instance, in the geographical case, most of the content accessed by a node was published by nodes that are located in a close proximity.

3.2 Design

As the focus of this thesis is tightly related to the Protocol Labs's IPFS ecosystem, it is important that our design is targeted at the principal research topics mentioned in [33] and in Section 2.5.2. After the study of the IPFS ecosystem and measurements that utilize DHT crawls as described in Section 2.5, we decided that our main focus was going to be targeted at minimizing the number of hops and the number of routing messages necessary to reach content and participants, within a defined concept of locality. The design, unlike the ones in [13, 45, 18] that are applicable only to geographical localization, should allow for an abstract concept of locality. For instance, as different applications are deployed on top of IPFS (e.g., databases, streaming services, file sharing platforms, etc.), it could be important for nodes running the same application to be able to reach that application's content faster, by having a lower number of hops between them in the DHT's protocol. Another example is using the geographical location. Nodes located on the same geographical zone (e.g., country, state or even city), might want to access content published by nodes within that same geographical region (e.g., documents, local websites, etc.) more often, which might take several unoptimized hops in DHTs that use a uniformly distributed identifier scheme to place nodes and content in the network. For the rest of this chapter, we will use the geographical notion of location to ease the explanation.

In order to achieve the goal of minimizing the number of hops in an environment where most of the queries have an access pattern associated with a given sense of locality, we reached the conclusion that one way to do it is to bias the identifier space so that nodes closer geographically are also closer in the identifier space. The Soft Partition scheme is this solution, which creates a bias in the identifier space by prepending the original node's identifier with a label that represents its localization. Another way of biasing the identifier space is to divide the nodes into different, smaller and hence, low diameter DHTs so that local content is available in a lower number of hops. The Hard Partition scheme captures this latter option, which creates a horizontal multi level DHT, building a new DHT for each of the labels. To enable communication between DHTs, an indexer is used that contains points of access to every DHT.

The label's genesis will depend on the concept of locality used. For example, with the labeling based on the application running on top of the node, each application would have to create a unique label for itself and every node would have that label in the bootstrap configuration. In the geographical sense of location, nodes could use a GeoIP service and match its approximate location with a configuration file in the bootstrap. The geographical location can be flexible and not be tied to any specific country or city. For instance, the world could be divided into multiple equal sized areas with each area having a different label. The labels also allow for a hierarchical division of participants and content, as the most significant bits can represent a larger area and the following bits representing subdivisions of that larger space. For instance, this can be used to represent continents that are subdivided into the multiple countries, and when needed, these can also be subdivided into densely populated cities or states. In practice, this hierarchical labeling will position identifiers under the high level areas closer together and so on, which allows for a more fine-grained optimization of the identifier space. We now discuss with more detail both approaches.

3.2.1 Soft Partition

The Soft Partition scheme enables the creation of multiple virtual partitions in the DHT's identifier space, by prepending to the original identifier a label encoding the node's geographical region. In classical DHT systems such as Chord [40] and Kademlia [30], nodes when entering the system are assigned a random identifier, which is uniformly distributed across the identifier space. This label, by becoming the most significant bits in the identifier, causes a shift in the participants position on the identifier space, placing nodes with the same label closer together. In the aforementioned DHT systems, nodes' routing tables will be filled with more nodes that have the same or closer label to their own, due to the way that the filling mechanism works. When publishing resources, publishers will also prepend their label to the content's identifier so that it will be stored under nodes in a nearby area. In Figure 3.1 this design is represented in a simplified manner, using a ring, where two bits are used to build the prefixes for the three geographical regions.

This modification causes a decrease in the number of hops necessary to communicate with peers and retrieve content with the same label (i.e., content that was published by a node with that same label), while also lowering the probability of communicating with geographically distant participants when retrieving content topologically closer, which in our system model, will have a higher cost due to the added latency.

Additionally, participants might have different usage patterns, which do not comply with the locality pattern (e.g., accessing content from some popular region). To improve the performance in those cases and to enable shortcuts in the identifier space, nodes are also able to cache contact information of participants with different labels, which will be managed outside the DHT routing table's management mechanisms. This cache is filled opportunistically during the regular searches and requests received by nodes. When searching for identifiers, nodes can use those entries together with the ones stored in their routing table to augment the amount of information known, and reach the searched identifier in an expected lower number of hops. These cached entries will be stored with a TTL. If the entry is not used by the end of the TTL or fails to respond to a query, it is discarded. The caching mechanism prioritizes nodes that are in the system for a longer period of time, so if an entry keeps responding to requests, the entry is kept, otherwise it is discarded.

In DHT systems like the ones previously mentioned, this modification can be relatively unobtrusive as the core protocol remains the same. The main modification this scheme requires is the prefixing of the identifiers, which could be useful for already long-running systems like IPFS as it would provide a smoother transition (e.g., nodes and new published content could have two identifiers while the scheme transition is in progress).

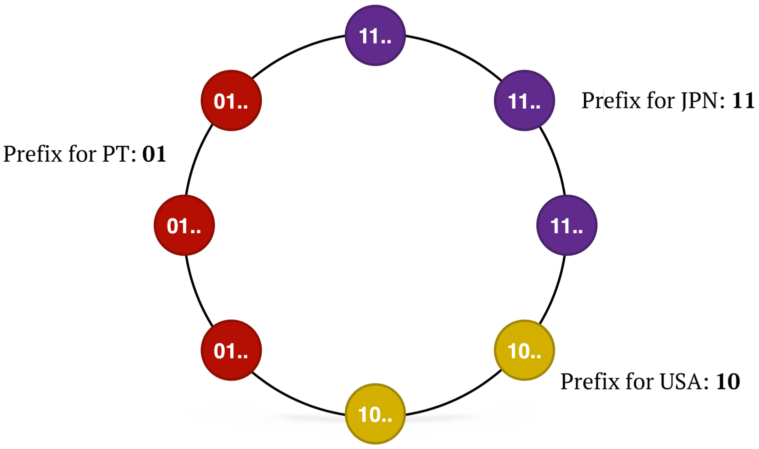


Figure 3.1: Soft Partition simplified design.

3.2.2 Hard Partition

The Hard Partition scheme induces the creation of multiple disjunct partitions by each one of the different prefixes. Each of the partitions forms a lower diameter DHT, which enables local searches (i.e., inside the local partition) to reach all participants and content in a lower number of hops. In this scheme, nodes will only add to their routing tables other participants that share the same label, as to maintain the separate DHTs. Unlike Soft Partition described in Section 3.2.1, the label does not have to be prepended to the node's identifier, it can be added to every message's and node's metadata. Content will also contain the owner's label information and as so, it is only required that it is published under the local DHT, as participants that want to retrieve content will only reach out to nodes with the same label as the content.

To enable search procedures in remote partitions, nodes rely on an external service that we will call index service, which acts outside the DHT protocol. The index service contains points of contact to other DHTs and acts as a bridge between them. This indexer can operate in a centralized way where there is only one instance of the indexer in the network, or in a decentralized way where different nodes know different indexer services with different points of contact, without affecting the correction of the DHT protocol. In Figure 3.2 this scheme is represented, using the same labels as in Figure 3.1, where each one of the labels create a separate DHT.

The index service design consists of a key-value table containing the multiple labels

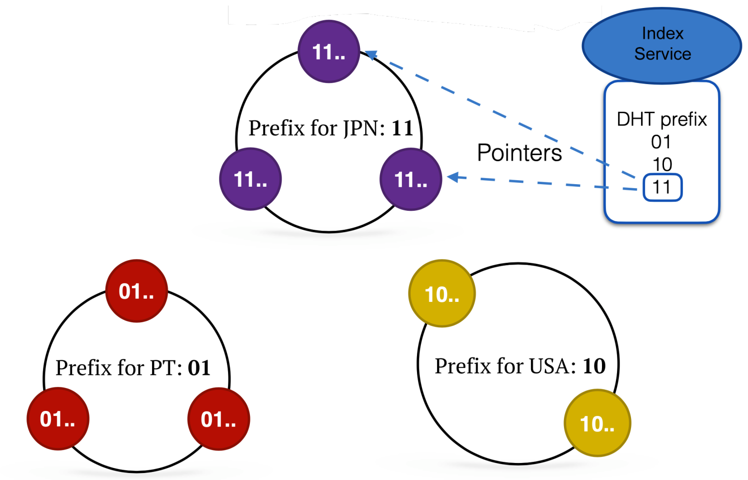


Figure 3.2: Hard Partition simplified design.

and some defined maximum number of contact points to each one. To diminish the possibility of only having stale entries to a certain partition, each contact point must be discarded after a defined TTL. If during this time the node is seen again by the index service, this TTL is refreshed. As these contact points will be used by nodes in other partitions, they will receive all the incoming traffic and can become overloaded. In order to mitigate this effect, contact points are periodically replaced by other nodes with the same label.

When joining the network, a participant must connect to at least one instance of the index service to retrieve the contact information for nodes with the same label, in order to join the correct DHT. The indexer address, like the contact information of the bootstrap node, must be in each node's configuration. After obtaining these bootstrap nodes, the rest of the joining procedure is dependent of the underlying DHT protocol used. If no node is known with the same label by the index service, a new DHT will be created and the new labeled entry will be added to the indexer.

As mentioned, there can be indexers scattered throughout the network working independently of each other. This is possible due to the only requirement the set of indexers must guarantee is that they know at least one node for each existing partition. To try to maintain this property, when entering the system, nodes must connect to at least one index service used and known by all other nodes. Nodes discover new index services during the regular operation of the network, as search messages when sent-out, can contain the list of indexers known by the initiator. Upon receiving a message with an unknown indexer, a node checks the index's status by performing a query to its own partition in order to create the label entry if not already present.

When in possession of contact information for multiple indexers after the bootstrap, nodes will prioritize the indexers with a better average response time, which is measured with the queries performed, thus only reaching out to distant indexers when the ones nearby do not have any valid entry for the requested label. This creates a load balancing effect between indexers with each one handling requests for different zones of the network.

To reduce the overload on the indexers, nodes cache a random subset of the entries received similarly to the one mentioned in Section 3.2.1. To lower the possibility of all nodes having the same contact points cached for a given label, every node periodically with a configurable probability, will send a refresh message to a random known index service, which will replace a random entry for the corresponding label.

3.2.3 Discussion

Both of these solutions serve the goal of reducing the number of hops to reach content and other peers that belong to the same locality group. The Soft Partition scheme is the less intrusive design to the underlying DHT functioning, as it only needs to change the nodes' identifiers in order to create the main desired effect of biasing the identifier space. In contrast, the Hard Partition needs to create changes to the routing protocol in order to find content outside a node's local DHT as it requires the indexer use.

The Soft Partition design has a great benefit compared with the Hard Partition, which is content caching. Due to the Hard Partition restriction of only storing on nodes that belong to the content's original DHT, smaller DHTs with a huge surge in its content's popularity might get overloaded as hot spots might emerge. On the other hand, in the Soft Partition, as queries can travel within multiple partitions until reaching the target, they allow nodes to cache the content along the query's path reducing a probability of hotspots for very popular content.

The Hard Partition has a benefit over the Soft Partition, which is that the access to every external partition is at the same hop distance, as it only needs to contact an index service to retrieve the correct contact points. This is useful for usage patterns that might access multiple partitions in a more uniform manner.

Comparing both solutions' design, the Soft Partition scheme can be more churn tolerant when interacting with external partitions, depending on the base DHT used. Due to the Hard Partition necessity of having index servers with available nodes for each one of the DHTs, in very unstable networks with the use of just one indexer, the information can become quickly outdated. To counteract this effect, it might be useful to use different indexers as they own independent network information between them (i.e., different nodes for each label), which increases the probability of having at least one point of contact available for a particular DHT. On the other hand, the Soft Partition relies on the underlying DHT's mechanisms to handle highly dynamic networks.

The indexers' uptime can also be a major factor in the Hard Partition churn tolerance. If a node only knows a single indexer, and it becomes temporarily unavailable, the node relies only on the peers stored in cache to communicate with other partitions. We designed this indexer with the IPFS infrastructure in mind, as they have multiple gateway nodes available that allow machines that do not run an IPFS node to interact with the system, which are expected to be always available and could also serve as reliable indexers for this design. Conceptually, any node can host an index with the downside of an increase in incoming traffic that can cause a weaker node (e.g., bandwidth, processing power, etc.) to become a bottleneck.

3.3 Implementation

Prototypes of both solutions were developed in the Go programming language and the DHT chosen as the foundation for our prototypes was the Kademlia DHT. This DHT was chosen due to not only being the DHT used in the IPFS system, but also due to the way the routing table management mechanism works, as a participant prioritizes peers that have an identifier closer to their own using the XOR metric described in Section 2.3.1.1. We now detail the implementation of both solutions, and the main challenges encountered.

3.3.1 Overview

The Kademlia baseline implementation follows the specification in the original paper [30] that includes which Remote Procedure Calls (RPC) should be implemented, the routing table main ideas, the XOR metric and the parameters that regulate each peer's fundamental behavior, which are:

- **K**: Regulates the maximum number of peers each bucket in the routing table has. It also defines the number of peers sent in the *FIND_NODE* and *FIND_VALUE* messages. In practice, this parameter impacts each node's out-degree and the overall network diameter.

- α: The alpha is the concurrency parameter. It defines the maximum number of queries that are sent in parallel during an RPC. While a higher value of α usually produces faster query resolution times, it also introduces extra load in the network as there is an increase in the number of messages used in a single round.

- β: While not explicitly mentioned in the paper, the authors mention a parameter that allows a node to wait for the reply of just a subset of the α messages and advance to the next round of messages. In the IPFS and other Kademlia based systems, this parameter is defined as β. By not waiting for the whole set of responses, the average resolution time can improve, but the overall query's efficiency can lower, as it can result in a greater number of messages.

- **Content Caching TTL** Defines the amount of time a node stores content received from a *STORE* RPC. For popular content, a larger amount of time is useful as it is expected that a large set of nodes will hold the content as cached entries are retained for longer, thus reducing the average hop count and query resolution time. On the other hand, content that is not popular might not be ever accessed in cached entries, and cause additional storage load.

- **Refresh time** States the interval of time between content's refreshing in the closest nodes to the content's key using the *STORE RPC*. A high frequency of refreshes minimizes the content lookup probability of failure on networks with high churn, while on stable networks can cause non-needed additional overload. This refresh can also be used for a node to refresh entries in its routing table's closest bucket, in order to update its surroundings.

3.3.2 Components

Our implementations follow an event-base model, in which there is a main loop within each peer that receives and processes messages, similarly to a state-machine. This model was partially based on the work in [7]. This means that while there is concurrency in the network, the message processing within each node is sequential. The transport protocol used to transmit messages between nodes is UDP, which was chosen due to being the protocol used in the original Kademlia's paper.

All three implementations (i.e., Kademlia, Soft Partition Kademlia and Hard Partition Kademlia) share an interface that exposes the four RPC and a hashing service used to create identifiers. The Soft Partition and the Hard Partition are built from the same components as the baseline Kademlia, but do not share the same event loop, as they require different message processing events and slightly different RPC implementations, which we will outline next.

Hashing service To create uniformly distributed identifiers, the SHA-1 hashing algorithm is used to create identifiers with 160 bits by receiving a string that must be unique to each node (e.g., IP and Port, randomly generated number, etc.), similarly to the identifier description present in the Kademlia's paper. Each of our solutions receives the label, and using the hashing service, modify their own identifier and the identifiers of the content they will store afterwards.

The Soft Partition uses the label received as an integer and prepends it as an array of bytes to the already hashed identifier. The Hard partition scheme for simplicity saves the label in the metadata of every node and content stored.

Routing Table The routing table is a key component of the Kademlia DHT. In the original paper the authors represent the routing table as a tree composed of multiple buckets, where each one stores contact information for nodes in a certain XOR distance range. This tree is unbalanced as it creates more buckets for distances closer to the owner's identifier. Our routing table is implemented using an array, where each position *i* has a bucket that stores nodes that have an identifier with a sequence of at least *i* bits in common with the owner's one, starting from the most significant bit.

Algorithm 1 Kademlia Routing Table

```
State:
   self
   buckets
   replacementCache
   K

Upon Init(self,K) do:
   self ⟵ self
   buckets ⟵ { }
   replacementCache ⟵ { }
   K ⟵ K

upon addOrUpdateNode(node, nodeId) do:
   bucket,bucketIndex ⟵ getBucket(nodeId)
   If node ∈ bucket then
      updateNode(bucket,node)
   elseif #bucket < K then
      addNode(bucket,node)
   elseif bucketIndex = #buckets-1 then
      Call splitBucket(bucket)
      Call addOrUpdateNode(node,nodeId)
   Else
      Call addToReplacementCache(node,nodeId)

Procedure splitBucket(bucket)
   Call createNewBucket(buckets)
   tmp ⟵ { }
   tmp ⟵ tmp ∪ getNodes(bucket)
   foreach node ∈ tmp do:
      (bucket,bucketIndex) ⟵ getBucket(nodeId)
      if bucketIndex = #buckets-1
         Call addNode(buckets[bucketIndex],node)
         Call removeNode(bucket,node)
```

Algorithm 1 describes the *addOrUpdateNode* function that is responsible for managing the routing table's state. This function is called every time a node receives a message from a peer and during the processing of a message if it contains peers' contact information in the payload. The function starts by determining the appropriate bucket the node belongs to, using the node's identifier and checking the length of the longest prefix in common with the owner's identifier. If the node already belongs to the bucket, it will update the last time the node has been seen. Otherwise, if it does not belong to the bucket, and it is not full, it adds the node to the bucket's list. If the bucket is full and is the last bucket (i.e., the bucket with

Algorithm 2 Kademlia Routing Table

```
upon getBestKNodes(key) do:
    bucketsSearched ⟵ { }
    nodesSeen ⟵ 0
    (bucket,bucketIndex) ⟵ getBucket(key)
    bucketsSearched ⟵ bucketsSearched ∪ bucketIndex
    reverse ⟵ False
    res ⟵ getNodes(buckets[bucketIndex])
    if #res ≥ K then
        return sortK(res)
    foreach index ∈ [1,#buckets] do:
        if reverse = False then
            next ⟵ bucketsSearched[index-1]+1
            if next = #buckets then
                next ⟵ bucketsSearched[0]-1
                reverse = True
        else
            next ⟵ bucketsSearched[index-1]-1
        bucketsSearched[i] ⟵ next
        res ⟵ res ∪ getNodes(buckets[next])
        if #res ≥ K then
            break
    return sortK(res)
```

the closest known nodes to the identifier), the bucket's split procedure is triggered and the *addOrUpdateNode* is tried once again. If none of those conditions are met, the node is added to the replacement cache that holds a limited amount of backup nodes in case peers present in the routing table stop responding.

Described by the *splitBucket* procedure, this function starts by creating an empty bucket and appending it to the routing table array. For each of the nodes from the previous last bucket, it checks if they belong to this new bucket. If a node belongs, it deletes it from the older bucket and adds it to the new one.

The method to retrieve the closest known neighbors to a given key is detailed in Algorithm 2. It starts by determining the bucket that holds the closest peers possible, adding it to a *searchedBuckets* list and its nodes to a *response* list. If this latter list already contains *K* nodes, it will return the list. If it does not contain the *K* nodes, and it is not the last bucket, it will iteratively start to go forward in the array and add the nodes to the *response* list. When the iterative process reaches the last bucket, it then starts going backwards starting from the first bucket added to the *searchedBuckets* list. The rationale behind going forward first is that nodes in the latter buckets will have at least the same longest prefix in common with the searched key. After all the necessary iterations, the nodes will be sorted by their distance to the key and only the best K nodes will be sent back.

Peers present in the routing table can fail, and the eviction policy implemented is similar to the one in the original paper. If a peer does not respond in a predefined amount of time three times, it will be removed from the routing table. Then, if present, an adequate peer from the replacement cache will take its place.

Procedures The messages used by the Kademlia's RPC and their processing had to be modified so that our solutions could be integrated. These modifications for the most part do not change the protocol's core behavior, except in the Hard Partition when the access to the index server is needed.

When a node initiates a *FIND_NODE* or a *FIND_VALUE* RPC in the Soft Partition, it begins by, if available, retrieving the known peers from the partition cache for that label, joining them with the closest K nodes from the routing table, and reducing them to the closest K nodes. Due to the limitation of only querying α peers at once, the closest α are queried while the rest is added to a result set used for later rounds, in case the first one does not find the searched key. The rest of the procedure follows the same behavior as base Kademlia.

A Hard Partition node's functioning for the aforementioned RPCs starts by checking if the label is equal to its own. If that is the case, then the rest of the procedure works in the same way as the base Kademlia. When the labels are different, which means that the searched key is in a different DHT, the protocol requests an index server for nodes with the required label. After receiving the contact points from the index server, the procedure works similarly to the base Kademlia. There is one additional case that needs to be addressed, which is to not add the peers received to the node's routing table during the procedure if the labels are not equal as to maintain the disjoint DHTs.

As a node might have multiple procedures running concurrently, their state (i.e., already queried nodes, nodes still not queried, number of replies since the beginning of the round, and the pending nodes) is saved in a dictionary for that RPC type, with the key being the searched identifier. If a new procedure for an already ongoing search is requested, the state between them is shared and the new request does not initiate a new RPC. During searches, while a number of replies (i.e., β) is not met or the list of pending nodes is not empty, the procedure will not advance to the next round, and will just keep adding nodes to the state. If by the end of a round, the procedure already queried the closest α nodes, meaning that the most recent round did not produce new closest nodes, or if the key searched is found in one of the responses, the procedure will terminate. The *FIND_VALUE* RPC can terminate earlier if a contacted node sends back the requested key and value. After the *FIND_VALUE* terminates, the initiator node can cache the key and value pair on the *K* closest nodes that have been previously queried but were not storing the pair.

In the *STORE* RPC, in our implementation and in the IPFS system, the content stored on other nodes is a pointer to the content owner's location (i.e., IP and port), while the actual content is only stored locally. This procedure starts by using the *FIND_NODE* RPC to discover the closest nodes to the content's key. When the procedure terminates, its state is used to store the key and value pair on the *K* closest nodes.

Refresh Events In the Kademlia original paper there are two main refreshing events. The first one is the buckets' refreshing, which is used to check the status of nodes in the bucket that have not been used for the longest. In our Kademlia implementation, we use this event to search for peers closer to the initiator node by searching the network for its own identifier. This use for the event generates more local knowledge about a node's close surroundings. Another effect of searching periodically by the own identifier is that in networks with high churn amounts, it helps nodes in maintaining its closest neighbors updated and add themselves to recently joined peers' routing tables.

The second event is used to republish keys stored in the network. As mentioned, keys stored on other nodes have an established TTL and after its expiration, the key is deleted. To keep the keys stored on the closest possible nodes, the content has to be republished before the TTL expires, to guarantee that it is available. In our implementations, nodes check if they possess a key stored by another node with an expired TTL in order to delete it. On the other side, nodes republish the keys that will expire soon with the same periodicity.

A third event required by the Hard Partition is the refreshing of entries in the Hard Partition. Due to nodes caching contact points from other partitions, if the entries were not periodically refreshed, all nodes would eventually have the same points of contact creating a bottleneck, as those nodes would have to process all the messages from external partitions. To try to counteract this effect, nodes periodically send refresh messages to the index server to offer themselves as a contact point to the network.

Index Server This is the main distinguishable feature of the Hard Partition and is integrated as an additional service that can run on top of any node in the network. In order for nodes to connect to this service, an index client was also implemented that manages the list of known indexers and the cached nodes that are gathered during queries to other partitions.

The index server implementation, similarly to the DHT implementation, is an event-driven model with a main loop that processes messages. The index itself is based on a dictionary that keeps a list of contact points to every label known with a maximum predefined size. The entries in these lists are updated regularly with the reception of *refresh* messages sent by index clients. If an index server receives a query from a node in a new partition, an entry in the dictionary is created, with the node becoming the first value stored under it.

The index client receives *query* messages from the DHT and relays them to an index server. To minimize the time spent waiting for server responses, two methods could be employed. The first and simpler method is to relay the queries to all known servers and wait for the first that responds with contact points. This method might produce the fastest resolution times but can also cause additional overload to the servers, as all nodes query all servers. The second and more sensible approach is to choose just the one index server, which has the lower average latency of all known servers. This latency is calculated using response times of queries made to the index servers. If a queried index server responds with an empty message, the next best index is used. This process repeats until all known index servers have been queried or one responds with contact points. The second message type the client handles are *refresh* messages. To reduce the possibility of server overload, this message is only sent with a defined probability to a known index server at random.

As mentioned previously, the client also caches entries sent by the index server, using a dictionary with the key being the entries' label. To each of the entries received, a TTL is added in order for nodes to periodically refresh their contact points, if not used for long. Before the client sends a *query* message, it first checks the cache for nodes with the requested label. During this check it only gathers the ones with a non-expired TTL while the others are removed. If the gathered set length is below K, it sends the *query* message to the index server and the cache is filled once again.

3.4 Summary

In this chapter we presented the first and second main contributions of this thesis. First, we provided an overview of the two topology biasing schemes created to build more efficient paths for queries that have an access pattern associated with a sense of locality. We detailed the differences between the Soft Partition and the Hard Partition and discussed their differences and benefits.

We then moved on to the second contribution, which is a concrete implementation of both schemes when applied to the well known Kademlia DHT, the challenges and main modifications that had to be done in order to support our schemes.

In the next chapter, we detail and discuss the experimental evaluation of our work.

4 Evaluation

In this chapter we present the experimental work done to evaluate our work. The main goal of this evaluation is to verify that our solutions have a better performance compared with Kademlia, when using similar usage patterns encountered in IPFS. Moreover, we want to check how our solutions and Kademlia behave in additional scenarios. With that purpose in mind, we begin by describing the considerations behind the choice of model for our underlying network (Section 4.1), followed by the experimental setup (Section 4.2), and finally a discussion of the obtained results (Section 4.3).

4.1 Network Emulation

Prior to the solutions' testing it was necessary to create a realistic network, in which the DHT could be deployed. As there is a scarce amount of information about the current state of the internet focusing on P2P interactions, it is hard to predict what a realistic network looks and behaves like. In this section, we discuss some tools considered and how, in the end, we created the network model used in our experiments.

4.1.1 Network models

In order to choose the network model, we studied different tools that allowed the creation of topologies with thousands of nodes. However, upon some experimentation with these tools, we found out that they did not offer what we required (e.g., different latencies between nodes, configurable network sizes, classification of nodes in different regions, etc.). We are now going to give an overview on some of the tools that were taken into consideration:

GT-ITM The Georgia Tech Internetwork Topology Models allows the creation of synthetic network models with some configurable options (e.g., number of nodes, number of edges per node). However, upon some experimentation, the networks created were overly random, with no resemblance of a real network, so this idea was abandoned.

Inet This tool is a generator for Autonomous System topologies[44] that can be used to create networks with a configurable amount of nodes. The team behind Inet used BGP routing table data, gathered from servers, to represent 51 Internet topologies from 1997 to 2002, which are used to model synthetic network models. This tool, even though it seemed promising, did not allow for the creation of small network models (i.e., less than 3037 nodes), which was needed due to emulation requirements that we will discuss further ahead in Section 4.2.

ARK IPv4 Dataset This dataset provided by CAIDA's ARK project [42], was created using 2.75 million addresses gathered by 40 monitors deployed in 24 different countries during the course of two weeks in 2019. The monitors periodically sent probe messages to a random IPv4 address and retrieved metrics such as the hops to reach the destination, the RTT from the intermediate steps until the destination, and geographical location of addresses.

As this dataset seemed the most promising, we processed the data in order to create a network graph connecting most of the probed IP addresses (i.e., the ones measured more than once with geographic data attached). The end result was a graph with 160 000 vertices, each representing an IP, and 260 000 edges, each representing a latency between two IPs.

Unfortunately, this dataset possessed some caveats that did not allow us to proceed with this choice for network model. One of them was that latencies were biased by the probing monitor, which means that probabilistically, nodes that are closer to the monitor had better latencies. With this in mind, we decided to use the distance between nodes as a heuristic to build synthetic latencies, which did not produce reasonable results as a high number of IPs had the same coordinates. Additionally, we noticed that there existed a relative higher density of nodes near the monitors' location. In the end, the plan of using this dataset had to be abandoned due to the aforementioned difficulties and to an unclear way of resizing the network to fit our emulation requirements.

4.1.2 Network creation

After experimenting with the datasets and tools mentioned in Section 4.1.1, we opted by creating our own network model. To this end, we used a Python module[1] that allows the creation and manipulation of graphs. A base network of 2000 nodes and another of 5000 were created with the following method: the desired number of nodes were positioned in a 2D plane with each axis representing a [0,1] interval. Every node pair that is closer than 0.15 distance units, is connected by an edge. To create a more realistic network, where a node has many close links and few distant ones, some close distance edges are switched for a random long distance edge with a probability of 0.1. In order to model the different

[1] https://graph-tool.skewed.de

network partitions, in which to base the labels, we resorted to a stochastic block model that creates a configurable amount of communities based on the existing edges' density between nodes.

To obtain the latencies between each node pair, we decided to use the distance between them, so we calculated the shortest path between the pairs, adding up the edges' weight. To create latencies that would make sense in a more realistic scenario, we multiplied the shortest path values by 1000 in the 2000 node network, and by 400 in the 5000 network, which will be explained in Section 4.2, forming a matrix with the latency values in milliseconds. Table 4.1 shows the average latency between nodes inside their local partition and to every other node in remote partitions for 2000 nodes, and Table 4.2 for 5000 nodes. As the number of partitions increases, each one being composed by a smaller number of nodes, the average local latency gets progressively lower, since the nodes belonging to a single partition are closer between themselves. On the other hand, the average latency to remote partitions remains stable, with a tendency to get closer to the value of the single partition as the number of partitions increases.

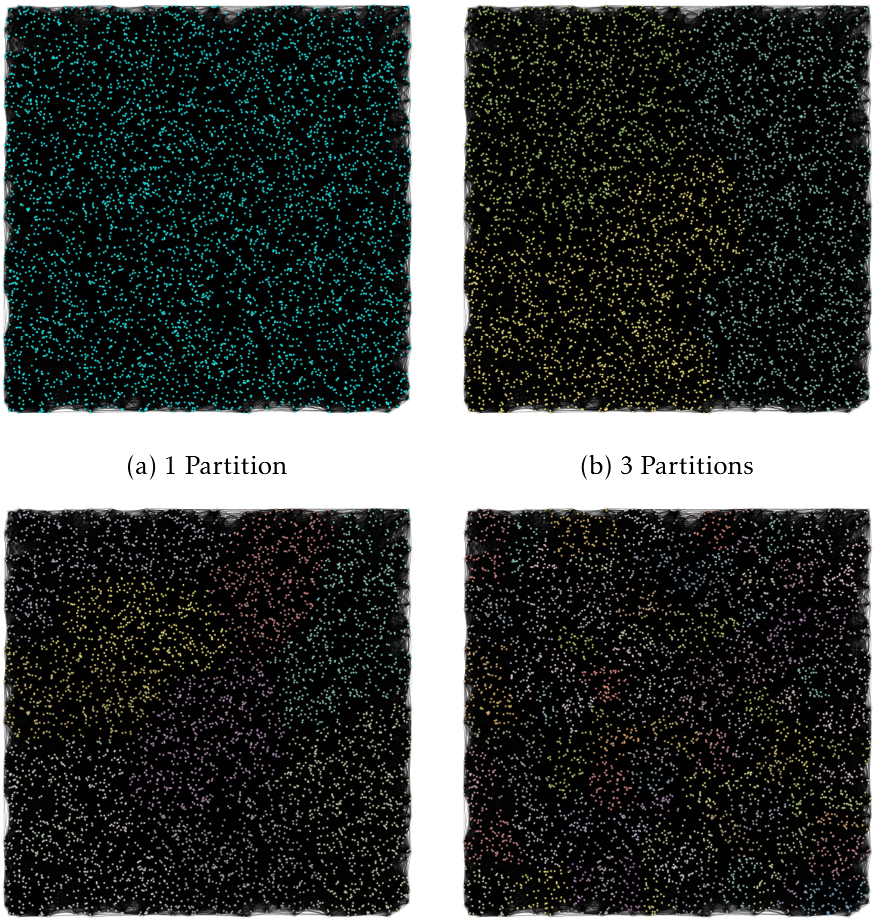
Table 4.1: Network average latency and partition sizes for 2000 nodes

Graph	**Local (ms)**	**Remote (ms)**	**Partition Size**		
			Avg	**Min**	**Max**
Single Partition	*1056.88*	*1056.88*	*2000*	*2000*	*2000*
3 Partitions	*662.70*	*1264.37*	*666*	*557*	*840*
5 Partitions	*533.44*	*1194.35*	*400*	*314*	*517*
10 Partitions	*364.01*	*1139.76*	*200*	*137*	*290*
100 Partitions	*91.39*	*1047.22*	*20*	*9*	*35*

Table 4.2: Network average latency and partition sizes for 5000 nodes

Graph	**Local (ms)**	**Remote (ms)**	**Partition Size**		
			Avg	**Min**	**Max**
Single Partition	*417.89*	*417.89*	*5000*	*5000*	*5000*
3 Partitions	*275.52*	*495.77*	*1667*	*1254*	*1519*
5 Partitions	*242.52*	*465.08*	*1000*	*701*	*1235*
10 Partitions	*145.03*	*449.68*	*500*	*358*	*744*
100 Partitions	*43.82*	*418.26*	*50*	*26*	*102*

Figure 4.1 illustrates the same network with 5000 nodes with a different number of partitions. For example, in Figure 4.1c, 10 partitions were created, with the different colors representing a network partition, and therefore the nodes' labels.



(a) 1 Partition

(b) 3 Partitions

(c) 10 Partitions

(d) 100 Partitions

Figure 4.1: Created networks with different partitions.

4.2 Experimental Setup

To run our experiments we used the platform *Grid5000*[2] that offers high performance machines. For the 2000 nodes network we used 4 machines and for the 5000 nodes network we used 20 machines, each one with an *Intel Xeon Gold 5220* and 96 gigabytes of RAM. The processes that ran our prototypes were equally distributed by 100 Docker containers, divided by all the used machines.

To apply the latencies previously calculated, we used the Linux *tc* tool that applies wait times to the delivery of every packet, based on the destination IP and Port. As each node has a different latency to every other node in the latency matrix, the original idea was to use a unique wait time for every destination. However, due to technical limitations imposed on the maximum number of different wait times by the tool used, this was not possible. To work around this limitation, we reduced the amount of unique latencies by grouping the ones with the same whole value, ending up with 2688 different latencies for

[2] www.grid5000.fr

the 2000 node network and 2754 different latencies for the 5000 node network.

Another limitation, imposed by the Linux *tc* tool in our early work, was that we could only scale the network to reach 2000 different processes, due to each process having to create a different rule for each IP Port pair. This grows exponentially as we increase the number of processes, and eventually starts to occupy most of the RAM and CPU computing power. In later work, we optimized the method by which a packet is routed through the network interfaces, allowing us to scale up to 5000 nodes. Additionally, we had to reduce the latencies values used in the 5000 node network by 60%, as the wait times imposed were overloading the CPU, which is why the latencies in Table 4.2 are lower.

When launching the processes to form the network, the contact point of all nodes is the first one that launches. For the Hard Partition's index servers, the first n processes that launch will become the n index servers. Figure 4.2 shows the location of the first node entering the system, which is circled in red to the middle left part of the graph.

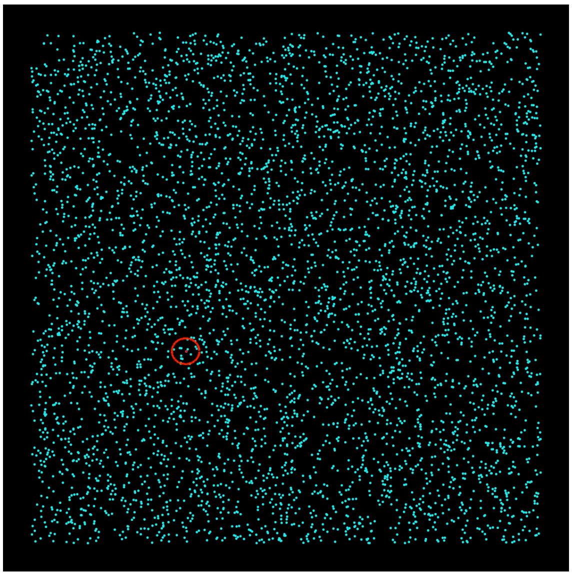


Figure 4.2: Index Server location.

4.2.1 Workloads

To test our solutions and compare it to the baseline Kademlia in different scenarios, we created different workloads for both the *FIND_VALUE* and the *FIND_NODE* RPCs. In each experiment, nodes have lists of all the available keys (i.e., one with all nodes' identifiers and other with the stored values' keys) with the attributed label. The workloads created to conduct our experiments were the following:

Biased Workload Our solutions' goal is to improve performance in workloads that assume a bias in content access towards locality. As so, this workload tailors the searches to find keys that match the searching node's label with a programmable probability.

Uniform Workload Each node chooses a random key from the set of available keys.

Zipf Workload We wanted to test the performance of our solutions when dealing with various popularity degrees. This workload applies a popularity degree to each of the partitions (i.e., labels) and to content. In IPFS and other P2P systems alike, there is some content that is more popular than others. However, in the IPFS case, as they serve various types of content such as static web pages or even resources for standard web pages, there is also a degree of popularity applied to the publishers' region (e.g., the USA news' web pages will have more global access on average than Portuguese web pages).

In *FIND_NODE* RPC experiments, after every node launches, we give 2 minutes of grace period for the network to stabilize and after that every node executes queries for 5 minutes using the keys the workload provides in intervals of 10 seconds. In the starting procedure, for the experiments using the *FIND_VALUE* RPC, we also give 2 minutes for the network to stabilize. Afterwards, each node stores 5 keys in the network and after 1 minute since the end of the store procedure, each node begins to search for keys in the network in intervals of 10 seconds.

In each query, we retrieve metrics such as the number of hops a query takes, by creating a graph with the nodes that were used during the query, and connecting the nodes in a message's payload to the nodes that provided them. We also retrieve metrics such as: the overall resolution time, which is the time span between when the query initiated and when the key was found; number of Kademlia rounds; number of contacted peers; the number of messages used; and from what partition is that key from. The *FIND_VALUE* RPC records the time and hops to find the contact information for the key being searched, plus the time to retrieve the content itself from the provider. In the Hard Partition we count the request to an index server as an additional hop in the query's path.

4.2.2 Experimental Parameters

In the experiments shown in Section 4.3, we varied different parameters that change the nodes' behavior. Besides the parameters detailed in Section 3.3.1, which are shared by all the prototypes, the solutions created use other important parameters tightly related to the functionalities of our solutions.

The Soft Partition only has two additional parameters in order to regulate the cache that stores nodes from external partitions. The *cacheSize* determines the maximum number of nodes stored in the cache by partition. If this parameter equals 0, then a node only relies on Kademlia's routing table to route messages. To periodically refresh this cache, the *cacheTTL* determines the maximum amount of time a node is valid without being used.

Regarding the Hard Partition, there are two types of parameters. The first type is associated with the index client and there are three parameters. Similarly to the Soft Partition, the index client also possesses a cache that is used to store peers received from the index server that has the same functioning and parameters. The client's last parameter is the *refreshProbability* that determines the probability of a node offering itself to be a contact point to its partition. The second type, which is associated with the index server, called *indexSize*, regulates the maximum amount of nodes stored for each partition.

4.3 Experimental Results

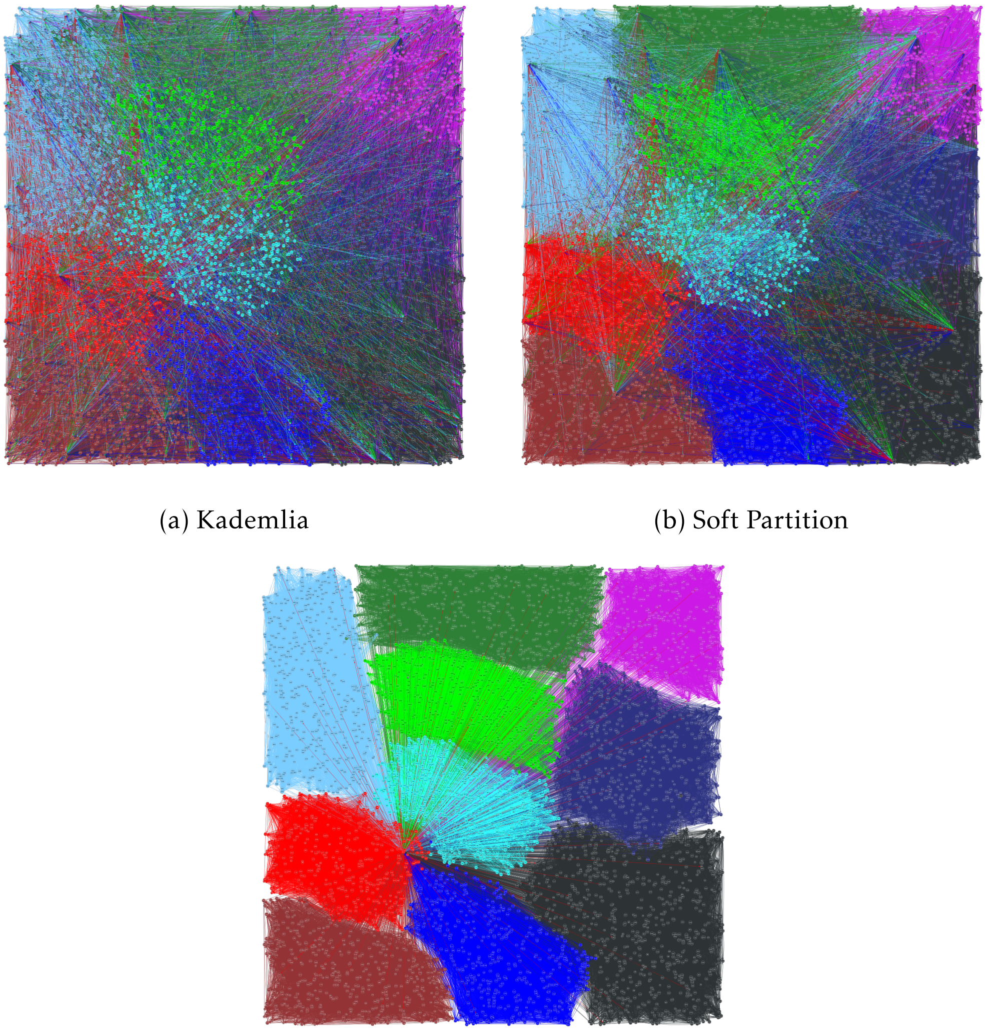
In this section, we present our experimental results. The overall objective of this evaluation is to compare both of our developed solutions to the base Kademlia in equal workloads. Additionally, we want to verify that the DHT parameters impact the different systems in similar ways.

We start by doing an analysis and comparison on the topologies each solution creates with the baseline Kademlia, then we move on to the discussion of the results under different workloads. Afterwards, we show the impact that each Kademlia parameter has on our solutions, and finally we compare the solutions with the baseline Kademlia under a churn scenario.

4.3.1 Topology Analysis

Before proceeding with the experimental results, it is important to discuss and analyze the topological changes our solutions apply to the overlay. To accomplish this, we deployed the 5000 node network, as described previously in Section 4.2.1, configured with 10 partitions. Every node performed the *FIND_NODE* RPC with a uniform workload, to have a higher probability of filling the routing table with diverse nodes. Afterwards, the nodes' routing tables were collected in order to construct a graph where nodes form unidirectional edges to every peer in their routing tables. The only parameter that influences the number of peers in the routing table is K, which is configured to 3.

Figure 4.3 depicts the visual representation of the formed graphs created by the DHTs, with every vertex having a color that encodes its partition, and the edges having the same color as the vertex they originate from. Figure 4.3a shows the baseline Kademlia DHT, which does not take the identifiers' labels (i.e., the nodes' partition) into consideration and as such, the connections each node makes are uniformized across the whole network. On the other hand, Figure 4.3b, which shows the Soft Partition graph, has more internal connections between nodes with the same label. In this graph, it is also noticeable that some vertices are more popular as they have more incoming edges from other partitions. This happens since nodes that join the network first are more likely to become a part of a node's routing table. Figure 4.3c shows the Hard Partition graph without caching



(a) Kademlia

(b) Soft Partition

(c) Hard Partition

Figure 4.3: Overlays created by the different solutions.

enabled. In this graph, the index server vertex establishes an edge with all its entries, and all vertices establish an edge with the index server.

Tables 4.3 and 4.4 show a set of graph properties from each DHT with 5000 nodes. We differentiate between the solutions' configurations without and with caching enabled (3 nodes for each partition). In the solutions with the index server, the number of nodes stored in the index are also taken into account, which are 10 nodes for each partition. We also distinguish between the overall properties of the graph and the properties when only viewed between nodes that belong to the same partition. The metrics we analyze are the following:

Out-degree Is a metric that captures the number of outgoing connections a node makes, which in practice is the number of nodes kept in the routing table plus the nodes stored in the cache. A high out-degree might hinder a node's performance in a

DHT by having to maintain a great amount of connections concurrently, while a low out-degree can be detrimental to the DHT's performance under churn.

Shortest path Measures the number of edges between a pair of nodes in a graph. The average shortest path is calculated by averaging all node pairs in the graph. This metric is correlated with the diameter of the overlay and is advantageous to maintain this value as low as possible, to ensure that a node can search the whole identifier space as fast as possible.

Clustering coefficient Informs how well-connected a node's neighborhood is by calculating the number of neighbors shared by a node and all of its neighbors, divided by the maximum possible number of shared connections. The average clustering coefficient is calculated by averaging the metric across all nodes. In a DHT, zones with a high clustering coefficient might be isolated easier, as nodes belonging to them do not create connections to a variety of nodes from different regions. In the tables, this metric was only applied to the overall topology due to the necessity of artificially separating the partitions, in order to calculate it individually, which can result in incorrect values.

Table 4.3: Graph Properties with 10 Partitions

	Out-Degree		**Shortest path**		**Clustering coefficient**
	Overall	Local	Overall	Local	Overall
Kademlia	29.42	3.1	3.55	3.96	0.37
Soft Partition	29.22	17.17	3.50	2.38	0.36
Soft Partition with Cache	52.55	17.19	2.71	2.38	0.21
Hard Partition	21.38	19.49	3.32	2.25	0.21
Hard Partition with cache	54.03	19.51	2.65	2.25	0.08

Table 4.4: Graph Properties with 100 Partitions

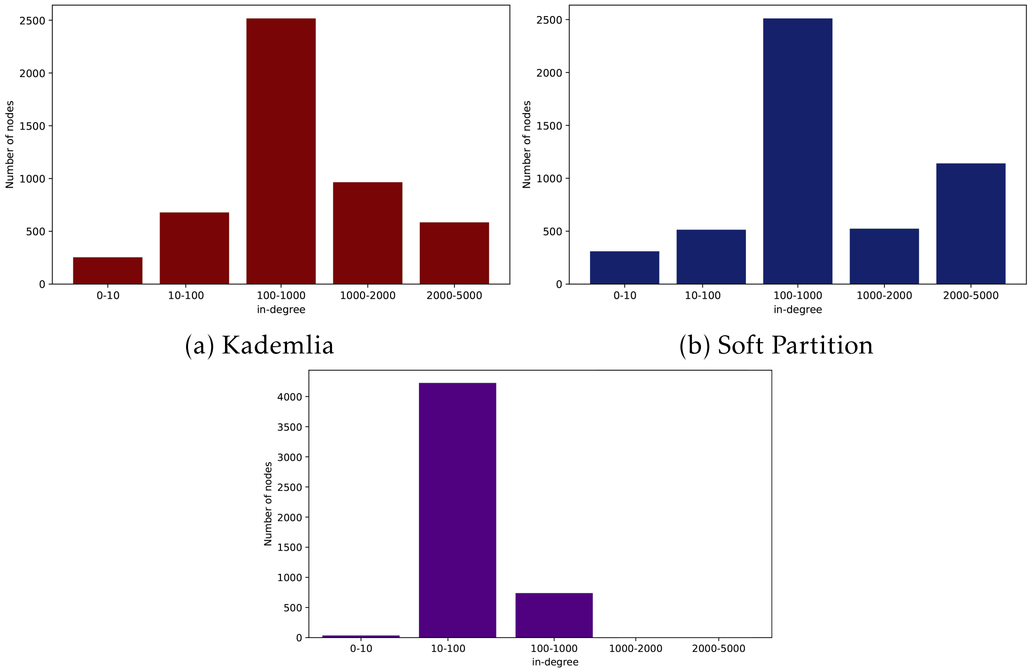
	Out-Degree		**Shortest path**		**Clustering coefficient**
	Overall	Local	Overall	Local	Overall
Kademlia	29.42	0.3	3.55	*3.91*	0.37
Soft Partition	33.86	13.01	3.41	1.85	0.34
Soft Partition with Cache	184.16	13.03	1.97	1.85	0.15
Hard Partition	15.34	13.15	2.58	1.79	0.41
Hard Partition with cache	197.4	13.15	1.96	1.79	0.06

Table 4.3 has the properties of the graphs created by the DHTs with the network configuration of 10 partitions. It is possible to observe that Kademlia and Soft Partition without using cache share similar metrics applied to the overall topology. The overall average out-degree of the Hard Partition is lower, and is explained by the routing tables only storing peers that belong to the same partition. Nevertheless, it still achieves a lower overall average shortest path compared to Kademlia as it utilizes the index server to reach distant regions of the identifier space. Even though a Soft Partition node stores, for the most part, peers from the same region, it achieves a similar overall shortest path compared to Kademlia as the few peers each node stores from outside regions seem to be enough to cover the whole identifier space. Where our two solutions improve the most is in the local shortest path, which decreases by more than one hop with this configuration of partitions. Both the Soft Partition and the Hard Partition with caching enabled have a similar out-degree that is much higher compared to the others. This is a result of each node storing at most 3 nodes per partition. The higher out-degree creates overall shortest paths in the network as it is more effective for each node to travel to distant zones of the identifier space. The nodes in the Soft Partition have the possibility of occupying space in their routing tables with peers from other partitions, which results in a slightly higher value in the local shortest path, which is also correlated with the lower value of local out-degree.

Kademlia and Soft Partition have similar average clustering coefficients, which might indicate a similar tolerance to churn. A reason for this relatively high clustering coefficient is that a large majority of nodes will have in their routing tables the peers that joined the network first. The Hard Partition experiences a decrease in this metric, since each node that joins the network uses different contact points, which are obtained through the index server and are periodically refreshed. The solutions with cache enabled have a much smaller value due to each node possessing a great variety of nodes in cache, obtained during the random searches performed.

Table 4.4 has the properties of the graphs created by the DHTs with the network configuration of 100 partitions. The Soft Partition retains similar properties from the overall graph with a moderate decrease in the overall out-degree, as nodes are less condensed in the identifier space, allowing them to fit into different buckets in the routing table. By increasing the number of partitions, the size of the partitions decreases and so, it results in a lower local out-degree, which can be seen in both the Soft Partition and the Hard Partition. An additional effect, which is a benefit for our solutions, is that the average local shortest path is greatly reduced when compared to the use of fewer partitions, and more importantly the baseline Kademlia. A less positive aspect of increasing the number of partitions is that in the solutions with caching enabled, nodes will eventually store a large amount of peers, if their search patterns are uniform across the whole identifier space. In these experiments as the nodes were searching for random keys dispersed by the identifier space, their caches from most partitions were not expiring, which results in a large amount of information that needs to be maintained. This latter effect is viewed by the large out-degree in both solutions. In workloads resembling the ones in IPFS, it is expected that the majority of queries will remain within the local partition, with some occasional accesses to popular content in other regions, which results in nodes retaining a small fraction of the cached entries.

A noticeable difference between the results reported on Tables 4.3 and 4.4, is that the Hard Partition's clustering coefficient increased almost by double. This is a result of each one of the 100 partitions having less nodes than with the use of 10 partitions and so, due to this less diverse set of possible peers, it is more likely that nodes in the same partition share a higher number of peers. On the other hand, when caching is enabled, the clustering coefficient has similar values compared with the use of fewer partitions, as each node creates connections with a diversity of peers from the different partitions. This same metric does not increase in the Soft Partition compared with the use of fewer partitions, as nodes have the possibility of connecting with peers from other partitions, making each one have a diverse set of links.



(a) Kademlia

(b) Soft Partition

(c) Hard Partition

Figure 4.4: In-degree histograms.

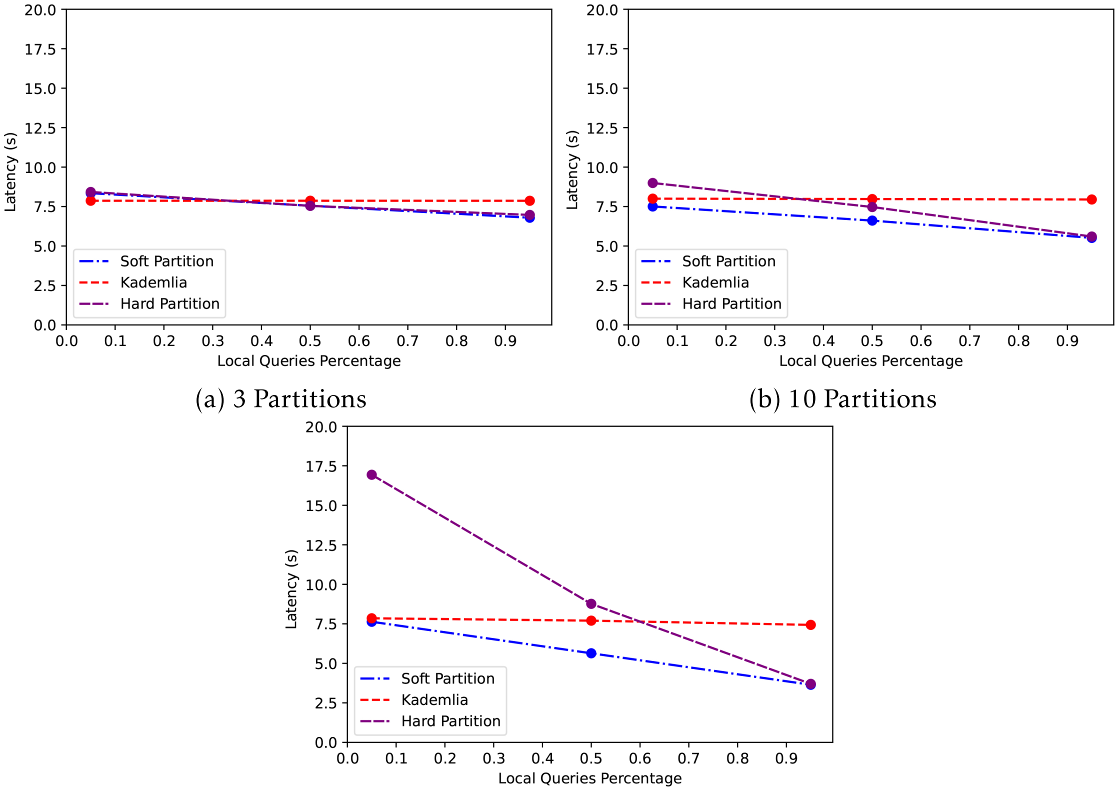
Figure 4.4 shows a set of histograms that represent the in-degree for all nodes in Kademlia and our two solutions without the caching of nodes enabled, using a 5000 node network with 10 partitions and a K value of 3. The results were collected from the same experiences as the ones in Table 4.3. We define the in-degree of a node as the number of nodes that have its contact information in their routing tables. We chose to represent the histogram with different bin sizes due to the large disparity between values, which made the uniform representation difficult to analyze. Starting with Kademlia in Figure 4.4a, the majority of nodes have an in-degree between 100 and 1000, while around 250 nodes have a degree between 0 and 10. In this last range, most nodes are the ones that were launched later, and due to most routing tables being already filled, only the nodes which are closer will insert them into their tables. There are also around 500 nodes (between 2000 and 5000), which are the ones who launched first that have a comparatively high in-degree. This last set may receive a large percentage of the queries, with a possibility of creating a performance bottleneck if the query rate received is too high. This popularity disparity can also be observed in Figure 4.4b that represents the results for Soft Partition. As it was seen previously in Figure 4.3b, there were some peers that became the contact points for a large percentage of nodes in other partitions, and due to node identifiers from external partitions fitting into the same buckets that are attributed to distant peers, these may be composed mostly of the first nodes that have joined the system. This difference in in-degree can also be attributed to all nodes having the same contact node when joining the system. As nodes use the peers received from the contact node to query and receive more information about their surroundings, these will also be the ones that are added to the routing table, thus creating a bias.

The Hard Partition, whose results are illustrated in Figure 4.4c, has a completely different range of in-degrees with more than 4000 nodes having an in-degree between 10 and 100. As different partitions form different DHTs, and as contact points are regularly changing in the index servers, even when nodes are still joining, the effect created by some nodes launching first is greatly reduced. However, there are still some nodes which have a very low in-degree, but the number is much lower when compared with the other two.

4.3.2 Biased Workload

With these experiments, we wanted to evaluate how well our solutions improved Kademlia over the assumed workload of a system such as IPFS (i.e., with more requests to nearby content). We expected that with a low number of requests to nearby content, all topologies would behave similarly, with Kademlia having a slightly better performance. On the other hand, as the amount of requests to nearby content increases, due to our solutions enabling nodes to know more peers from their partitions, we expected that a lower time would be needed to resolve a query, and the number of hops in the network to drop as well. Within this workload we varied the percentage of local queries by 5%, 50% and 95%.

The Kademlia parameters for these experiments were 3 for K, 2 for α, and 2 for β. The Soft Partition allows nodes to store 3 peers for each partition, with a TTL of 60 seconds, while in the Hard Partition configuration there is 1 Index Server storing 5 nodes for partition, while nodes store 3 peers for each partition with a TTL of 60 seconds. In the *FIND_VALUE* experiments, content caching after the retrieval is not enabled to not create an over-caching effect, which can happen due to the short time the experiment is running. Periodically, in intervals of 20 seconds, nodes query for their own identifier to refresh their closest bucket.



(a) 3 Partitions

(b) 10 Partitions

(c) 100 Partitions

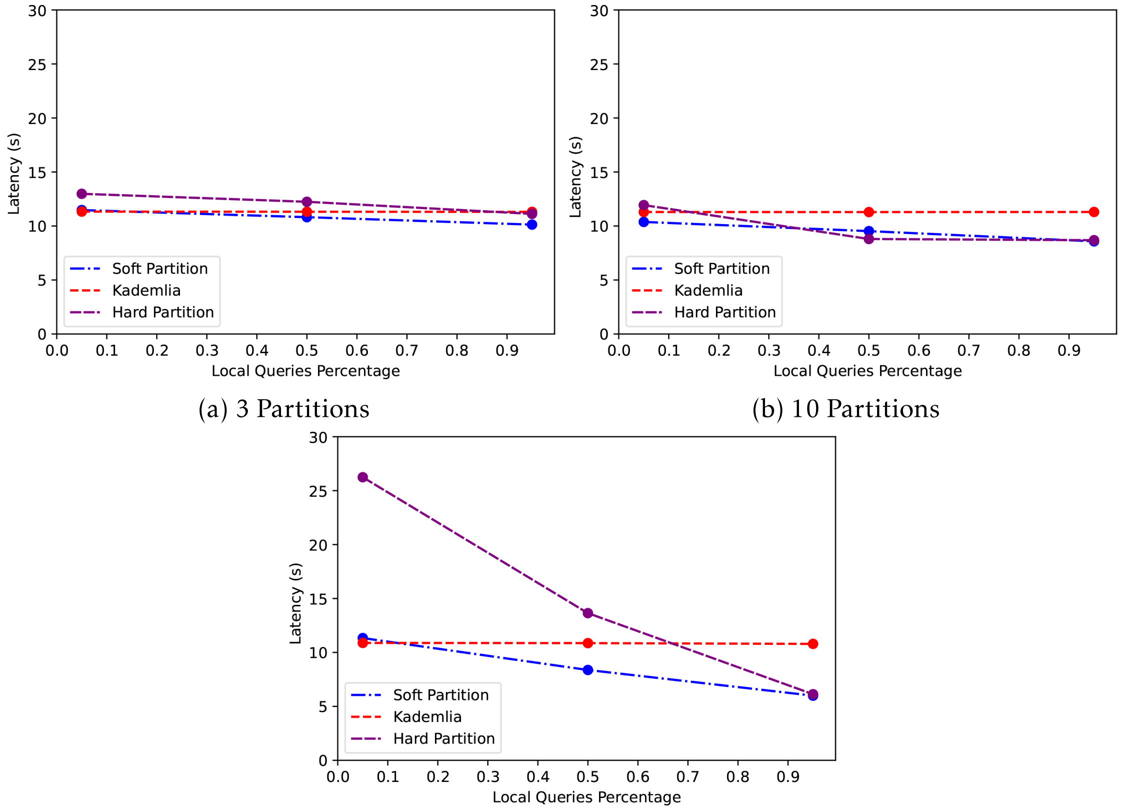
Figure 4.5: Average *FIND_NODE* query time with different number of partitions.

Figure 4.5 presents three plots, each one displaying the experiments using different partitions for the *FIND_NODE* RPC. On the X axis is the percentage of queries that are made to a key that shares the same label, and in the Y axis it is represented the time it took for a node to receive the requested key. With these plots it is possible to notice that Kademlia does not have a response to the locality of the requests, as the average time remains almost static for each setting. On the other hand, as the number of partitions and the locality of requests increases, both Soft Partition and Hard Partition are able to resolve queries in a lower amount of time.

In an almost perfect scenario, which corresponds to a 95% requests targeting local identifiers, both solutions reach an almost similar level of performance, which is expected since the graph properties, displayed in the previous tables, show that both solutions create similar topologies when looking solely at a partition level.

One interesting result regarding the Hard Partition alternative, is its performance illustrated in Figure 4.5c when using a low percentage of local requests, which is much lower than the other DHTs. This is explained by the use of just one indexer, which in this set of experiments became overloaded due to most queries requiring access to it. As there are numerous partitions, and every time each node wants to query a key with a label that is still not caching, or the cache has expired, must access the index server, it generates a high amount of inbound traffic to it. We later will discuss how the use of multiple indexes spread throughout the network makes a massive difference in the performance of the Hard Partition.

The Soft Partition, in Figure 4.5a with a low percentage of local queries has a slightly higher resolution time compared to Kademlia, which is explained by each node with a low number of partitions being able to fill most of the routing table with nodes from its own partition, and thus the average shortest path to external partitions increases slightly. The similar result by the Hard Partition can be attributed to nodes receiving a similar subset of nodes to external partitions when they start to query, which hinders their performance as they become slightly overloaded due to all external requests. This Hard Partition challenge can be managed by changing the caching policies or by increasing the amount of nodes the index server knows in each partition, in order to create more diverse caches.



(a) 3 Partitions

(b) 10 Partitions

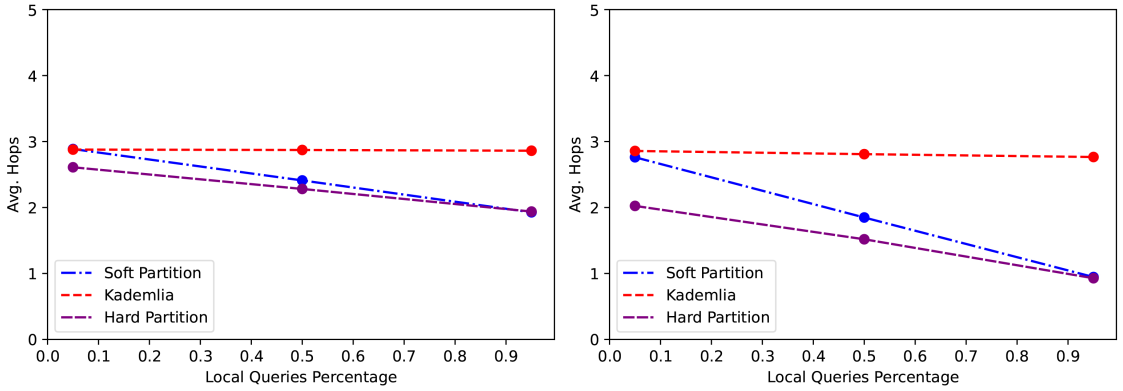
(c) 100 Partitions

Figure 4.6: Average *FIND_VALUE* query time with different partitions.

Figure 4.6 shows the same plot configuration but for the *FIND_VALUE* RPC. In these experiments, it is expected that the average resolution time increases compared to *FIND_NODE*, as it is necessary to find the key that contains the resource's location and then request it to the original provider. The overall results are very similar to the first set in Figure 4.5, with a more noticeable improvement in query resolution time when using a higher number of partitions and higher query locality. Figure 4.6b shows the Soft

Partition achieving better performance in all scenarios compared to Kademlia, while the Hard Partition is hindered by the index bottleneck in lower percentages of local queries. On the other hand, when the workload uses a query locality of 50%, the Hard Partition returns to the expected resolution times as there is a balance between the diversity of nodes in cache, as they expire more often, and the frequency of accesses to the index server, which will be fewer since half of the queries are directed at the local partition. In Figure 4.6c, the Hard Partition has, once again, a drop in performance when compared with the other two when dealing with low locality content, which is explained by the same reasons previously mentioned. However, with 95% local queries, achieves a similar performance to the Soft Partition.

In Figures 4.7 and 4.8 the number of hops for each of the RPCs is represented for 10 and 100 partitions respectively. As expected, both our solutions achieve a lower number of hops as the percentage of local queries increases. When queries are mostly local, both of our solutions have an identical average number of hops since the topological properties are similar. Due to creating shortcuts in the network, by the use of the index server and the caching of 3 nodes for each partition, the Hard Partition enables a lower hop count for external requests. The Hard Partition caching mechanism only requires a single request to an index server, maintaining fresh contact points more easily, when opposed to Soft Partition caching that, when expired, takes longer to fill due to using the nodes found during requests. Although the Hard Partition allows queries to resolve in a lower number of hops, the hop to the index server can be very costly as seen from Figures 4.5c and 4.6c.

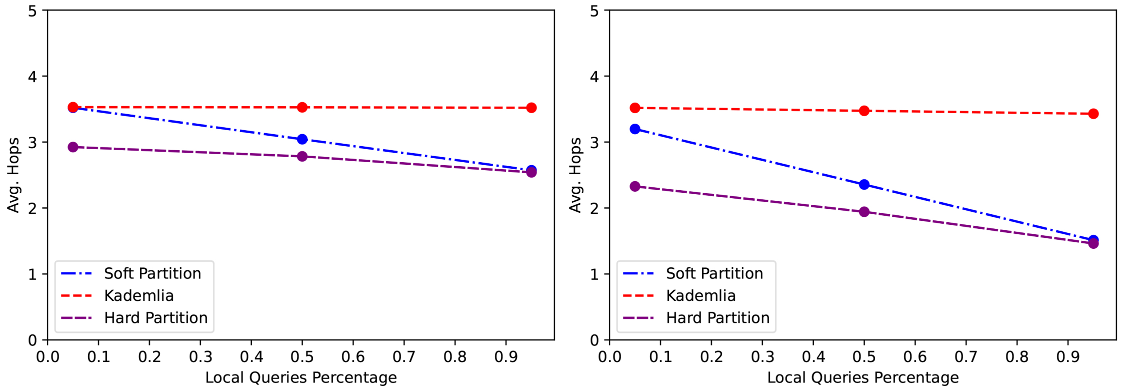


(a) 10 Partitions

(b) 100 Partitions

Figure 4.7: Average hops for *FIND_NODE* with different number of partitions.

To study how much of an impact the use of multiple indexes in Hard Partition does, we performed the same set of experiments using 1,2, and 3 indexes with 100 partitions. Figure 4.9 shows the results extracted from the *FIND_VALUE* experiments compared with the previously obtained data from Kademlia. In the Y axis we display the resolution time only for the set of queries made to external partitions, as these are the only cases where the use of the index server makes an impact. As previously seen, the use of only 1 index with a low number of local queries creates a performance bottleneck and the average time



(a) 10 Partitions

(b) 100 Partitions

Figure 4.8: Average hops for *FIND_VALUE* with different number of partitions.

increases noticeably. In this same workload setting, the Hard Partition's performance with a higher number of indexes is also affected by the large amount of concurrent queries, but due to the load being divided by the multiple indexes, it is less noticeable. As the amount of local requests increases, the indexes' load becomes increasingly smaller to the point where with 50% local queries, the use of 2 indexes can resolve queries in lower average times than Kademlia. With 95% local queries, the average times between the multiple index settings becomes negligible as the single one can handle every request.

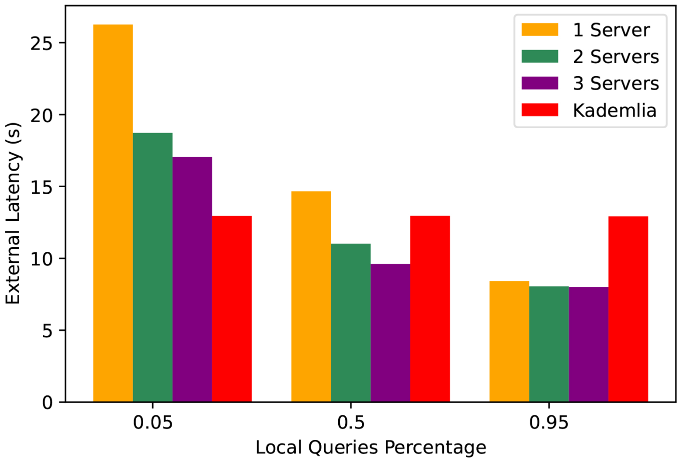
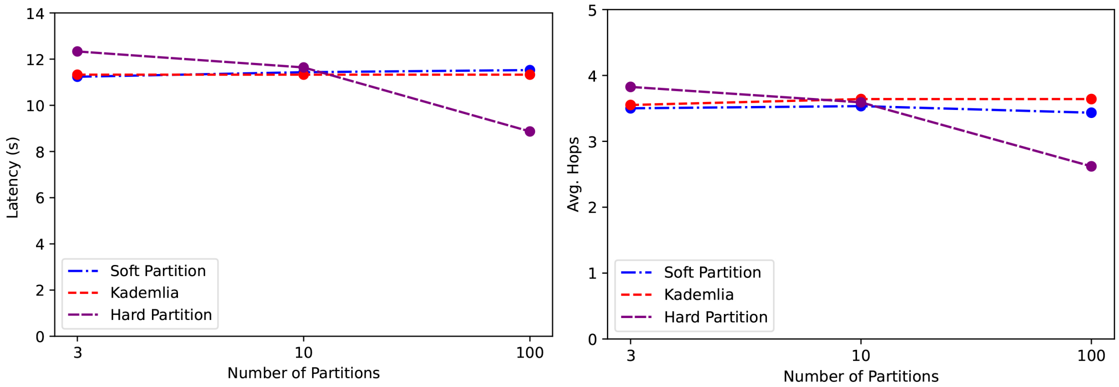


Figure 4.9: Average *FIND_VALUE* query time to external partitions with multiple Hard Partition indexes in a 100 partitions configuration.

From all of these results, we can conclude that in the network settings used and with the expected workload of systems such as IPFS, both Soft Partition and Hard Partition achieve the main objective by reducing the resolution time of queries with high locality. By the caching of nodes from external partitions, and the consequent increase in out-degree, nodes can also benefit from a small improvement when querying other partitions. It was interesting that Kademlia with all the different workload settings remained with similar levels of performance across the whole spectrum, as we were expecting that it could also show improvements in resolution time when dealing with high locality workloads, as the keys searched were kept in nodes with a relative low latency. As seen from the presented results, when the nodes rely heavily on a single index server in the Hard Partition, the performance can drop, which is why we allow multiple independent index servers to operate in the network, distributing the load. When using multiple indexes, nodes can also choose the one that offers the better performance, which also helps to reduce resolution times if the access to them is frequent.

4.3.3 Uniform Workload

The purpose of using a uniform workload is to guarantee that by biasing the Kademlia's topology, usage patterns that do not follow the expected workload maintain a similar level of performance when compared with the baseline Kademlia. The DHT parameters for these experiments remain the same used in Section 4.3.2, in order to keep the experiments consistent in their results. The only change in parameters is the number of servers used in Hard Partition, which is 3. This was changed in order to reduce the overloading possibility, and focus the experience on the topological changes.



(a) Average query time

(b) Average hop count

Figure 4.10: *FIND_VALUE* for different partition configurations.

Figures 4.10a and 4.10b, in the Y axis, show the *FIND_VALUE* results for the resolution time and the number of hops respectively, while the X axis represents the different number of partitions used. Figure 4.10a shows that the average resolution time from Kademlia and Soft Partition remained similar across the whole spectrum. The obtained result from Kademlia is expected, as it does not use the partitions' information. However, it is possible to see a minor increase in Soft Partition's resolution time as the number of partitions increases. These results can be explained by a majority of nodes possessing the same contact points to other partitions in their routing tables, which are the nodes that were the first to join the network. As the number of partitions increases, the searches performed will have more diverse destinations, which require communication with nodes from external partitions. Due to the routing table being composed mostly of nodes that belong to the same partition, there is a chance that a large group of nodes tries to communicate with the same set of nodes, thus slightly overloading them. Nodes in Soft Partition also cache entries to other partitions, however the TTL used seemed to not be enough for nodes to always have contact points available.

On the other hand, the Hard Partition starts with a higher resolution time and as the number of partitions increases, this value decreases. There are two related factors that explain the results exhibited by the Hard Partition scheme. The first one, is that a node in Hard Partition relies only on the peers sent by the index server to perform external queries, as it does not retain contact information about other nodes in external partitions. As nodes start to request content around the same moment, there is a high possibility of many nodes possessing the same points of contact to external partitions, which can become slightly overloaded and take longer to respond. Due to these nodes not failing completely, they are not removed from cache and are maintained for the duration of the whole experience. The increase in partitions diversifies the partition queried, which causes cached nodes to hit the TTL and be replaced by other contact points, thus minimizing this phenomenon. The other factor is that when querying external partitions, the nodes in cache can be very distant from the searched key, which might increase the overall hop count and consequently the resolution time. As the number of partitions increases, the diameter of each DHT becomes smaller, so this effect does not have as much impact in these cases. Figure 4.10b shows the average hop count for the different number of partitions, which remained similar for Kademlia and Soft Partition. This is a positive result as it may show that the topology biasing does not significantly degrade the overall query hop count for uniform queries. The explanation for the Hard Partition's suboptimal performance in a low number of partitions can be confirmed by this figure as the hop count is higher than the other two. However, with a high number of partitions this count reduces greatly as the use of the index allows a query to create shortcuts in the identifier space.

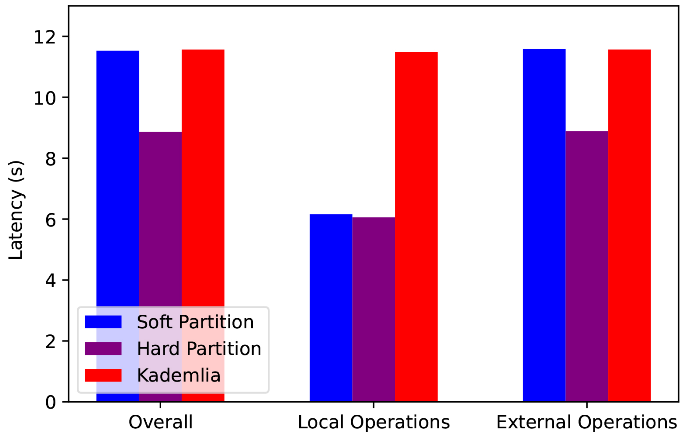
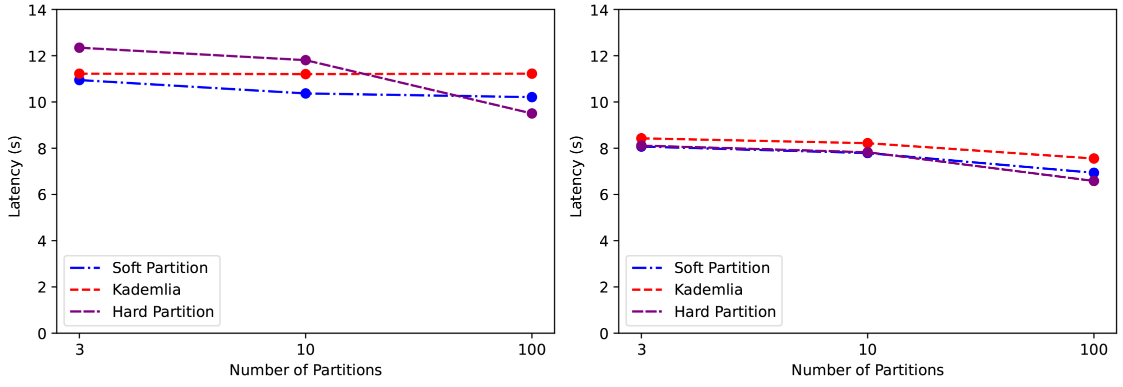


Figure 4.11: Average *FIND_VALUE* query time in a 100 partition configuration.

Figure 4.11 shows the experiment with 100 partitions more fine-grained, differentiating the queries that are local and the queries that are external. Our solutions achieve almost half the average time of Kademlia for local operations, while for external ones only the Hard Partition sees benefits compared to Kademlia. For Kademlia, it was expected that local queries would take less time to resolve, as nodes with the same label have lower latencies between them and the retrieval is expected to take less time. Still, the uniform identifier distribution does not seem to allow taking advantage of this condition. The Soft Partition, by caching nodes from the other 100 partitions, was also expected to gain a certain advantage over Kademlia that did not seem to occur, but as explained before, the TTL used was not enough to maintain nodes cached at all times. In the case of the Hard Partition, even with the TTL used being the same and the nodes in cache expiring, the use of the three indexes to refresh the cache and lower diameter DHTs allowed for faster resolution times.

4.3.4 Zipf Workload

This workload's objective is to verify how each solution behaves when dealing with content with a highly varied degree of popularity. For these experiments we used a Zipf of 1.01, and we distinguish between the results obtained with the use of content caching enabled and without it. The Zipf is not only applied to the published content but also to each of the partitions, in order to represent the more widely accessed regions of IPFS. Content caching in this context means that every time a node retrieves content, it maintains it for a TTL of 60 seconds and also sends a *STORE* operation, containing the content's identifier and the contact information for the original provider, to the closest *K* nodes that did not return the value.



(a) Caching not enabled.

(b) Caching enabled.

Figure 4.12: *FIND_VALUE* query time for different partition configurations.

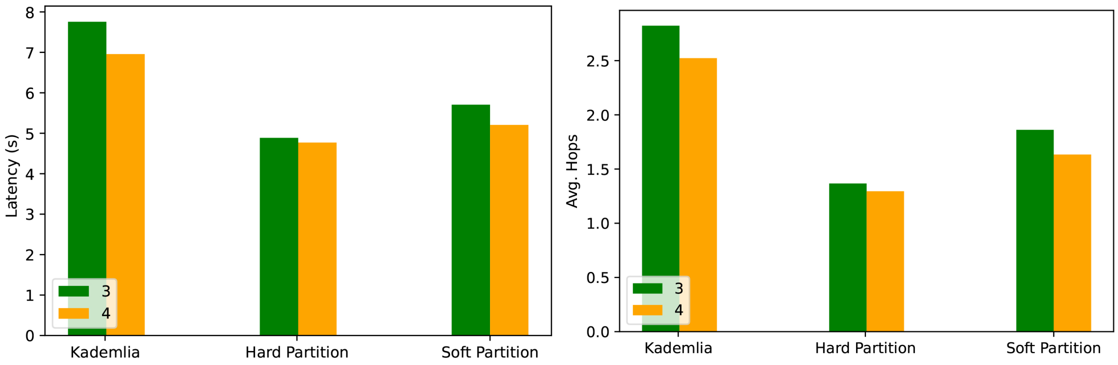
The plots in Figure 4.12 show the average resolution time for the two caching scenarios mentioned. The plot in Figure 4.12a, without caching enabled, shows the Soft Partition and Kademlia behaving similarly, with a small decrease in resolution time for the Soft Partition in the larger number of partitions, explained by the caching of nodes from external partitions. The Hard Partition in this plot starts with a higher average resolution time compared with the other two. One reason for this result is that, unlike Soft Partition, which caches nodes opportunistically during searches, this solution only stores nodes sent by the index server that, if not used for a while, are removed. With this Zipf workload there will be some partitions that are more popular than others, and nodes outside this partition will never let the contact points to them expire, as these are used often. This effect can be positive because the use of the index server is minimized, but as explained previously in Section 4.3.3, these contacts can be shared by many nodes, causing them to take longer to respond. Another side effect of the workload is that cached nodes from partitions with a low popularity will expire more often due to being rarely queried, which require the access to an indexer, increasing the resolution time. With a high number of partitions, the Zipf distribution creates even more disparity between the partitions' popularity thus reducing the probability of querying less popular regions. Additionally, as the formed DHTs have a very low diameter, it takes on average less hops to reach the target.

It is possible to extract interesting observations from Figure 4.12b, as all solutions seem to behave similarly when content caching is enabled. It was expected that the Soft Partition and Kademlia would have a better performance when compared with the Hard Partition, as nodes in those two DHTs are allowed to store content from other partitions. The reason for this similarly is that, due to the comparatively small amount of nodes composing each DHT, the popular content is stored in a large percentage of nodes, and these queries, for the most part, only need to query a contact point and then the original provider to resolve.

4.3.5 Kademlia Parameters

In this set of experiments we wanted to study how each Kademlia parameter (i.e., K, α and β) impacted our developed solutions individually. To do this, in each experience we fixed two of the parameters while varying the other one. For these experiments, we used a biased workload of 50%, in order to not only compare each of the DHTs individually, but also to check how the parameters affect the performance in the assumed access pattern targeted in this work. The RPC chosen in these experiments was the *FIND_NODE* as it is the simpler one to understand its behavior. Moreover, as the other RPCs use it as a basis, the effects noticed will be transversal.

The set of plots in Figure 4.13 show the experiments that varied K, while the α and β are both 3. In the two figures, each of the DHTs has a set of two bars with each color representing the value used for the parameter. Figure 4.13a displays the average amount of time that a query takes to resolve, and Figure 4.13b the average number of hops. As expected, the increase in K reduces the resolution time and the number of hops in all DHTs. These results are explained by the increase in out-degree, making each node have more information about the network, and can choose better which nodes to query first. A related factor is that when nodes are queried for a key, they return K peers, which also increases the amount of information that can be used by the querying node.



(a) Average query time

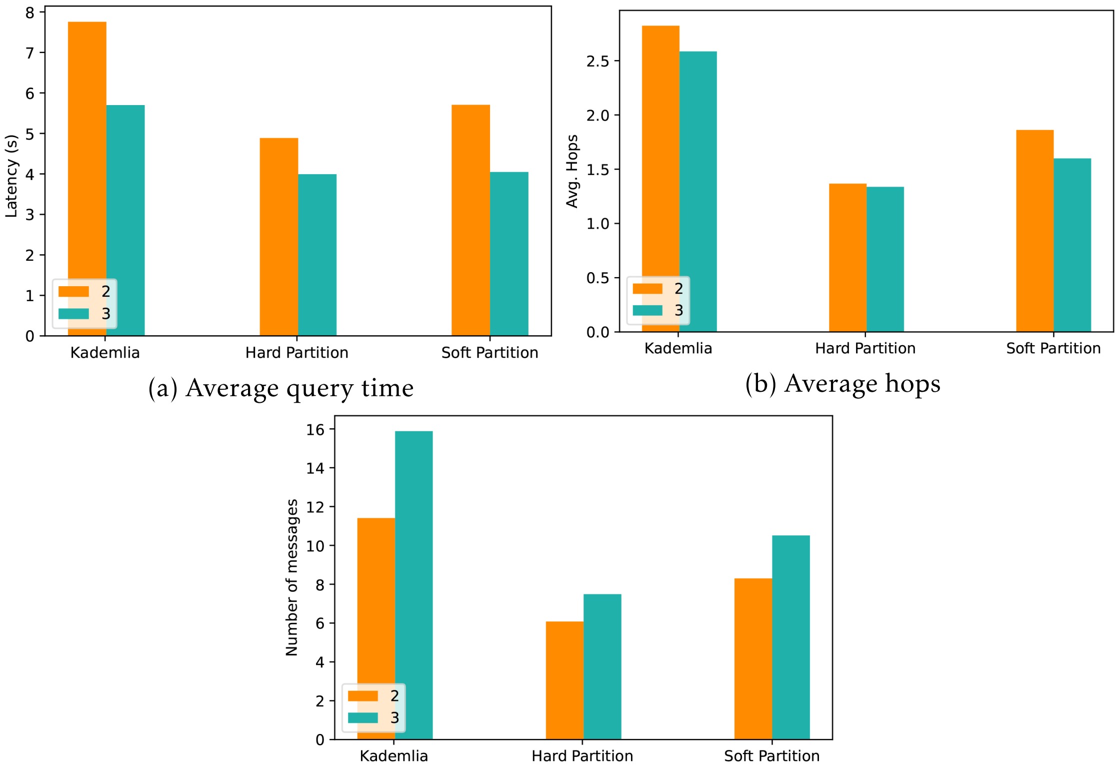
(b) Average hops

Figure 4.13: *FIND_NODE* varying K in a 100 partition configuration.

Kademlia has the higher resolution times and hop count with this workload but also gains the most performance from the increase in out-degree, as the workload is biased. From the tables previously seen in Section 4.3.1, a Kademlia node on average has a low number of connections to peers from the same partition, which increases as the buckets grow. Comparatively, a Soft Partition node has more than half of the routing table filled with nodes from the same partition, and an increase in K does not create such an accentuated increase in performance. With even less benefits, a Hard Partition node experiences only a slight reduction in hop count for this workload when the K increases, since it only fills the routing table with nodes from the same partition

The set of plots in Figure 4.14 show the results for the experiments with the same workload type, while varying α between 2 and 3. In these experiments, K and β were 3 and 2, respectively. In Figure 4.14a, by increasing the number of messages sent concurrently by 1, the average resolution time decreases substantially, as the amount of information produced about other peers in each round is higher. In Figure 4.14b, the average hop count decreases when we have a higher value of α, which can be associated with the search span increasing as it contacts more nodes at the start of each round, and the probability of encountering the request key in earlier requests is higher. This increase in search span can also be associated with the data reported by Figure 4.14c, that shows the number of messages exchanged on average for each search also increasing. Even if it may produce faster resolution times, the latter plot can show that a higher α can also be detrimental to the DHT function, as it can cause additional stress on the overall network.

In terms of percentage of the overall resolution time, Soft Partition and Kademlia have similar gains, with 30% lower resolution time. The Hard Partition does not see a large benefit, as part of the query time requires the access to the index servers. To achieve similar levels of performance, Kademlia uses a comparatively higher number of messages with a 40% increase with the parameter at 3. Once again, the Hard Partition requires a lower number of messages by using the index server to reach the desired partition directly.



(a) Average query time

(b) Average hops

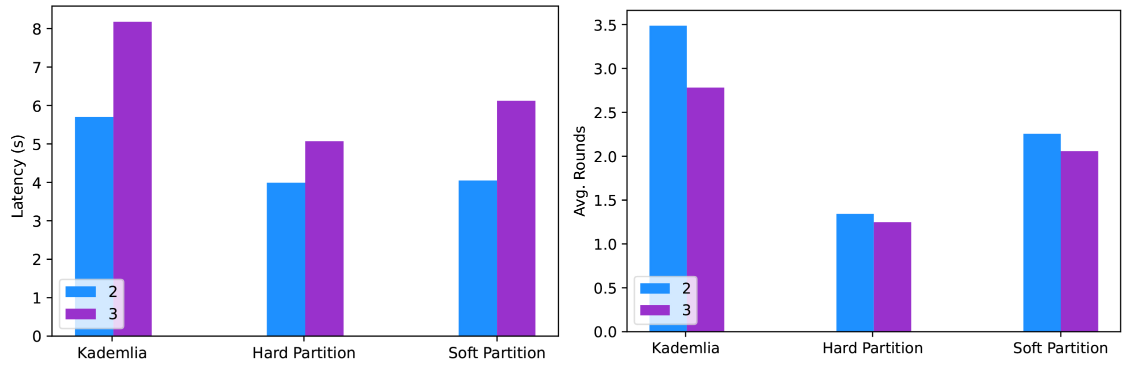
(c) Average messages

Figure 4.14: *FIND_NODE* varying α in a 100 partition configuration.

The last Kademlia parameter that we discuss is β, which determines how many responses are required to move to the next round of queries for a given key. In these experiments the K and α are both 3. In Figure 4.15a, which shows the average resolution time, all DHTs have a higher resolution time when the queries have to wait for more responses in order to advance to the next round. Figure 4.15b shows the average rounds needed in order for a query to resolve. The overall result is that all three require fewer rounds to find a searched key with a higher β, as a query at each round gathers more identifiers and will only progress to the next round if the key was not found. Both Soft Partition and Hard Partition only have a minor decrease in this metric, since these two solutions in the workload used will possess and gather information about the keys being searched more easily.

4.3.6 Churn

In these experiments we wanted to study how each of the solutions behaved when dealing with a large failure of nodes at the same time. In real large DHT deployments like IPFS, these types of failures can happen, if for instance there is a failure at the ISP level, or a new software version has an unexpected behavior that creates incompatibilities between versions. User patterns can also cause similar churn scenarios, for example in day/night patterns as users from certain regions turn off their personal machines.



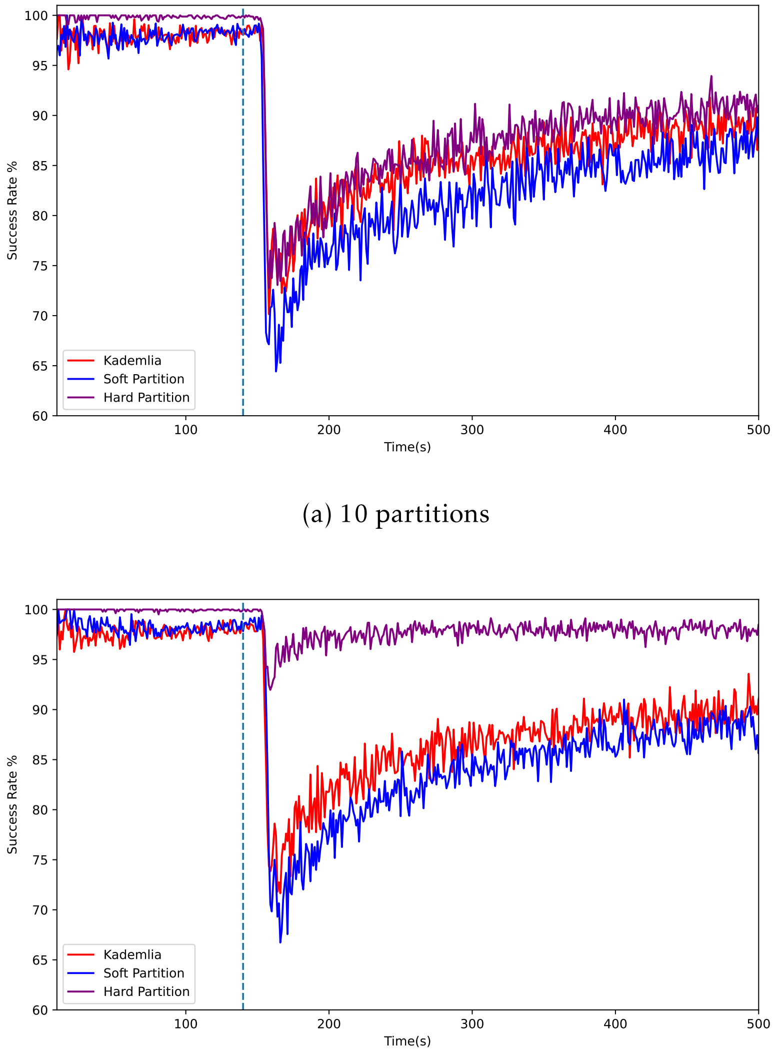
(a) Average time

(b) Average rounds

Figure 4.15: *FIND_NODE* varying β in a 100 partition configuration.

To simulate a similar behavior, we launched the 5000 node network with a uniform *FIND_NODE* workload that only uses node identifiers, which were present in the system during the course of the whole experiment. Two minutes after all nodes launched, we terminated a percentage of random nodes in order to see how the queries' success rate changed. The experience terminated 9 minutes after the last node launched. We collected the same metrics as the previous experiments grouped by the query's termination time rounded to the second. The DHT parameters used are the same as used in previous sections. There are other additional parameters to have in mind in these experiments. The first one is the *timeout time* that determines the amount of time a node waits for peers to respond to a query before marking the node as unresponsive, which is set at 8 seconds. We found that 8 seconds is flexible enough for overloaded nodes to respond while also not maintaining dead peers in the routing table for an excessive long time. If a peer is marked 3 times in a row as unresponsive, it is removed from the routing table. We chose to not remove a node straight away to allow for some flexibility, as highly popular nodes (e.g., the bootstrap node) may get sometimes overloaded, and take longer to respond. The second parameter is the *refresh time*, which not only sets the interval between searches of a node's surroundings, but also sets the interval between Hard Partition nodes sending *refresh* messages to the index servers. This refresh parameter is set at 10 seconds while the probability of a node sending one of these messages is 30%.

The plots presented in Figure 4.16 show the DHTs' success rate results with 10 and 100 partitions and a uniform node failure of 30%. Starting with Figure 4.16a, Soft Partition sees the largest drop in success rate, while the Hard Partition and Kademlia have a similar success drop. This result is interesting as it shows that a majority of nodes may rely on a similar small subset of peers to communicate with different partitions, which once this set fails, can cause nodes to lose a large part of their routing tables, and disconnect less popular nodes (i.e., with a low in-degree) from the wider network. This last hypothesis can explain why the network never fully recovers to higher success rates, as some nodes become unreachable. The large drop in Soft Partition can be attributed to nodes relying on the information transmitted by the bootstrap node when joining the network, which



(a) 10 partitions

(b) 100 partitions

Figure 4.16: *FIND_NODE* success rate with 30% instant failure.

has a bias towards nodes in its partition that when queried by nodes who belong to different partitions, may return a similar subset of nodes as their identifier matches the same routing table's bucket. The Hard Partition does not suffer as much from the same problem, as it is likely that nodes are presented with different contact options by the index server, due to the constant refreshes, when joining the network.

The most interesting results can be observed in Figure 4.16b with the use of 100 partitions. While Kademlia and Soft Partition have similar results between themselves, the Hard Partition appears to have a comparatively higher churn tolerance, which might be explained by two main factors. The first one is that by using a higher number of partitions, the DHTs created become more cohesive with a single node knowing a larger percentage of their DHT, which does not allow it to break so easily. The second and most important factor is attributed to the Index Server that when faced with an instant failure of nodes, it can recover rapidly by being refreshed with nodes that are alive, allowing external queries to resolve with success once again.

An additional result that we can extract is related to the first moments before the instant failure occurs. In the two figures, both Soft Partition and Kademlia do not reach a steady success rate of near 100% unlike Hard Partition, which may be attributed to a longer period of network stabilization as they form a larger single network. As previously seen, there will exist nodes that are more popular than others, which makes it harder to find less popular nodes as they are stored in fewer routing tables. To counteract this, nodes could use a higher value of K that will increase the probability of less popular nodes belonging to a routing table.

4.3.7 Discussion

From the analysis of the properties of the formed topologies, we could set a projection for the kind of performance our solutions would have compared with Kademlia. The designed solutions form topologies where a node's out-degree is mainly composed of nodes that share the same label, thus creating shorter paths between them. When the caching of nodes from other partitions is enabled, the overall shortest path is also greatly reduced compared with Kademlia, at the expense of a higher cost to maintain the peer information. The in-degree of nodes can also be a factor to consider in query performance as the routing table Kademlia offers prioritizes nodes that are in the system for longer, creating, in this way, unbalanced networks where some older nodes have a high in-degree, and may receive a large sum of queries, while nodes that join the system more recently may struggle to be added to routing tables. To minimize this effect, a larger K could be used, which allows younger nodes to be more easily added to routing tables, with the increase of contact management as a side effect.

We explored the use of different workloads and different parameters in our evaluation to better understand where our solutions stand out from Kademlia. As we increased the number of partitions, the differences between Kademlia and our solutions became more evident as the topology changes had more drastic effects. In biased workloads with a heavy percentage of queries directed to a node's label, both Soft Partition and Hard Partition greatly improved the performance of Kademlia by lowering the number of hops to the target keys in both *FIND_NODE* and *FIND_VALUE*. The number of indexes also plays a major factor in the Hard Partition's performance, as just one server might not handle the requests of all nodes in an acceptable time frame.

The similarities between our solutions and Kademlia in uniform workloads were a positive result, as we expected that the performance of our solutions would deteriorate in these environments. However, it is important to notice that both of our solutions had caching enabled, which allowed them to create shortcuts in the identifier space. Without this type of caching, all three DHTs would be more similar when querying nodes in different partitions.

Using the Zipf workload, which changes the popularity of regions and content, all three solutions seem to behave similarly, with better resolution times for the developed solutions when the nodes are divided across more partitions. When we allow nodes to cache content, all three solutions achieve lower resolution times as the average hop count is greatly reduced. This can lead to an over-caching effect if the content TTL is not properly regulated. As our experiments have a relatively short length, and the query rate required to have an accurate sample for all nodes is high, the over-caching effect can be hard to control, making highly popular content available in a large percentage of nodes, requiring only a couple of hops to reach the target.

The change in Kademlia parameters produced similar effects in all three DHTs, which is a positive result. The Hard Partition might achieve similar levels of performance than the other two with lower K values, since it creates smaller DHTs and the fewer slots in the buckets might be enough to guarantee a correct routing. However, K also helps in maintaining routing in unstable networks, which is the case of IPFS as it is composed of mostly personal machines that are disconnected regularly. All solutions benefit from a higher α, although this parameter has to be carefully chosen in order to not produce an unnecessary amount of messages that can compromise the overall system's performance.

From the experiments with a percentage of instant failures, we could extract that the Hard Partition might recover quickly from these scenarios, as the contact points stored in index servers have a high probability of being available. Those experiments also show that this solution reaches a steady state quicker than the other two, which are more prone to a success rate drop while subjected to churn. To target these types of environments, all of these DHTs could benefit from higher K and α, as it produces more expensive (and robust) searches. Additionally, a shorter interval between refreshes could also benefit the DHTs in those circumstances as it would eliminate unresponsive nodes faster.

Additional experimental results are present in Annex I, which show different settings (e.g., Kademlia parameters, partition's configuration, etc.) used in the conducted experiences.

4.4 Summary

In this chapter we presented the methodology behind our evaluation and its results. We first explained and detailed the creation of our network model, while discussing some considered options. We then detailed the experimental setup, the multiple parameters, and workloads used. Finally, we presented the results of the conducted experiments, discussed the results and the lessons learned from them.

In the next chapter we conclude this thesis by showing our conclusions and detailing possible future work.

5

Conclusion and Future Work

Conclusion

Distributed Hash Tables, by organizing participants predictably, provide P2P applications with efficient lookup protocols capable of surpassing some limitations of centralized solutions. However, classical DHTs assume that all participants have similar needs or behaviors, by placing participants randomly in the spectrum of possible identifiers, which does not apply to multi use P2P systems such as IPFS. As such, these systems can benefit from a non-uniform identifier space where participants that have similar behaviors and access patterns to content are closer together, minimizing the necessary number of network hops to communicate between them.

Before designing our solution, we discussed the challenges of building P2P overlays and studied the state of the art of structured and unstructured overlays, as both can be combined to provide better churn tolerance while not incurring high maintenance overheads. Afterwards, we presented the case study of IPFS, by detailing how it is built and what are the future challenges it is expected to face regarding scalability. With this study in mind, we created two alternative topology biasing schemes that by using generic labels, which can be provided in different ways (e.g., the node's application or a GeoIP service) and are attached to identifiers, promote a higher connectivity between nodes that share equal labels as they become closer. The Soft Partition scheme creates virtual partitions in the identifier scheme, grouping participants with equal labels together, while the Hard Partition scheme creates a multi level DHT, by building a separate DHT for each label, which are then linked through indexers that store contact information for arbitrary nodes in each one of the partitions. Moreover, both of our solutions are also designed with a caching system, inspired by the unstructured overlays, which enable participants to maintain information about peers in other parts of the network.

In addition to the schemes' design, we implemented both designs over the popular and widely used, Kademlia DHT that is also the basis for the IPFS system. We described the challenges encountered and the Kademlia's protocol changes required to enable our solutions to run as intended. Afterwards, in order to evaluate our solutions, we described the topology model, with different numbers of partitions and nodes created to emulate a realistic scenario, in which participants utilize different workloads and system settings, tailored to represent different behaviors.

The experimental results reveal that both of our solutions are effective in providing faster and more efficient lookup protocols when a high percentage of queries are directed at content stored in the local partition, as it minimizes the required hops. For content stored by participants that belong to external partitions, our solutions also enable queries to those parts of the network more effectively, by maintaining some additional state. With regard to churn tolerance, our results indicate that there is a tendency towards faster recovery from the Hard Partition scheme, providing that reliable indexers exist in the network.

Future Work

In this last section we detail possible future work to integrate into the design of our solutions and additional evaluation scenarios to better understand the ramifications of our work.

Index DHT In the Hard Partition's design, indexes work independently, and every index may hold contact points to every single partition in the system. In large scale networks with possibly millions of participants, it could be useful to deploy more index servers connected by a DHT of their own, and use the contact points as the objects that are stored in it. This not only provides storage load balancing between indexes, but also provides an efficient and faster way of deploying new indexers.

Dynamic cache allocation Both solutions could benefit from having a dynamic cache that adapts itself to the usage pattern of each participant. This cache would allow a node to cache more entries for partitions that queries more often, in order to allow for a more optimal choice of peers when required.

Dynamic routing table Inspired by another Kademlia version [17], it could be interesting to integrate a dynamic allocation with regard to the number of peers in each bucket. The first buckets (i.e., the ones with the more distant peers) would hold a higher number of entries, as it is more common to find peers that belong to them and would offer a more diverse set of peers from different partitions to a Soft Partition node. This size would get progressively smaller until it reaches K.

Label types Related with the evaluation, a possible setting that would be interesting to test the solutions, is to evaluate using labels that are not based on the geographical proximity, such as application labels that will make distant nodes belong to the same partition.

Heterogeneous Partitions In our experiments we used cluster settings, which create similar sized partitions. It could be interesting to represent more realistic settings, where there are some regions or applications, which have different numbers of participants, as it would allow the gathering of useful insights about the topological biasing schemes.

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 (cit. on pp. 2, 7, 20, 27, 28, 32).

[2] B. Beverly Yang and H. Garcia-Molina. "Designing a super-peer network". In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. 2003, pp. 49–60. DOI: 10.1109/ICDE.2003.1260781 (cit. on p. 12).

[3] N. Carvalho et al. "Emergent Structure in Unstructured Epidemic Multicast". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 2007, pp. 481–490. DOI: 10.1109/DSN.2007.40 (cit. on pp. 1, 8).

[4] B. Cohen. *The BitTorrent Protocol Specification*. 2008. URL: https://www.bittorrent.org/beps/bep_0003.html (visited on 11/30/2021) (cit. on pp. 2, 7, 20, 30).

[5] "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web". In: *Conference Proceedings of the Annual ACM Symposium on Theory of Computing* (1997), pp. 654–663. ISSN: 07349025 (cit. on p. 16).

[6] P. Á. Costa, P. Fouto, and J. Leitão. "Overlay Networks for Edge Management". In: *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*. IEEE. 2020, pp. 1–10 (cit. on p. 12).

[7] P. Á. Costa, A. Rosa, and J. Leitão. "Enabling wireless ad hoc edge systems with yggdrasil". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 2020, pp. 2129–2136 (cit. on p. 39).

[8] 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 (cit. on p. 7).

[9] 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 (cit. on p. 18).

[10] 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 (cit. on pp. 2, 23).

[11] 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 (cit. on p. 9).

[12] *go-ipfs*. 2021. URL: https://github.com/ipfs/go-ipfs (visited on 11/30/2021) (cit. on pp. 27, 29).

[13] C. Gross et al. “Geodemlia: A robust peer-to-peer overlay supporting location-based search”. In: *2012 IEEE 12th International Conference on Peer-to-Peer Computing, P2P 2012* (2012), pp. 25–36. DOI: 10.1109/P2P.2012.6335806 (cit. on pp. 25, 33).

[14] 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 (cit. on pp. 2, 20).

[15] S. Henningsen et al. “Mapping the Interplanetary Filesystem”. In: *IFIP Networking 2020 Conference and Workshops, Networking 2020* (2020), pp. 289–297. arXiv: 2002.07747 (cit. on pp. 27, 29).

[16] 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 (cit. on p. 10).

[17] R. Jimenez, F. Osmani, and B. Knutsson. “Sub-second lookups on a large-scale Kademlia-based overlay”. In: *2011 IEEE International Conference on Peer-to-Peer Computing*. 2011, pp. 82–91. DOI: 10.1109/P2P.2011.6038665 (cit. on p. 72).

[18] A. Kovacevic, N. Liebau, and R. Steinmetz. “Globase.KOM - A P2P Overlay for Fully Retrievable Location-based Search”. In: (2008), pp. 87–96. DOI: 10.1109/p2p.2007.18 (cit. on pp. 26, 33).

[19] P. Labs. *Multiformats*. 2020. URL: https://multiformats.io/multihash (visited on 11/30/2021) (cit. on p. 28).

[20] J. Leitão, J. Pereira, and L. Rodrigues. “Epidemic Broadcast Trees”. In: *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. 2007, pp. 301–310. DOI: 10.1109/SRDS.2007.27 (cit. on pp. 1, 7, 8).

[21] J. Leitão. "Topology Management for Unstructured Overlay Networks". PHD Thesis. Instituto Superior Técnico, 2012 (cit. on pp. 1, 5, 6, 8, 9).

[22] 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 (cit. on pp. 1, 6, 9, 13–15).

[23] 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 (cit. on p. 8).

[24] 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 (cit. on pp. 1, 9, 15, 16, 26).

[25] 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 (cit. on pp. 11, 16).

[26] 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 (cit. on p. 26).

[27] E. K. Lua et al. *A survey and comparison of peer-to-peer overlay network schemes*. 2005. DOI: 10.1109/COMST.2005.1610546 (cit. on pp. 1, 9).

[28] 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 (cit. on pp. 1, 9, 27).

[29] 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 (cit. on pp. 2, 10, 27).

[30] 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 (cit. on pp. 2, 3, 6, 10, 17, 18, 27, 29, 32, 34, 38).

[31] 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 (cit. on pp. 11, 25).

[32] 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 (cit. on p. 10).

[33] *Multi-Level DHT Design and Evaluation*. 2020. URL: https://github.com/protocol/research-RFPs/blob/master/RFPs/rfp-7-MLDHT.md (visited on 02/10/2021) (cit. on pp. 3, 33).

[34] J. Paiva, J. Leitão, 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 (cit. on pp. 2, 26, 27).

[35] 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 (cit. on pp. 10, 17).

[36] 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 (cit. on pp. 28, 30).

[37] 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 (cit. on pp. 1, 6, 7).

[38] *Routing at Scale*. 2020. URL: %7Bhttps://github.com/protocol/ResNetLab/blob/master/OPEN_PROBLEMS/ROUTING_AT_SCALE.md%7D (visited on 11/30/2021) (cit. on p. 31).

[39] 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 (cit. on pp. 2, 10, 27).

[40] I. Stoica et al. “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”. In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pp. 149–160. ISSN: 0146-4833. DOI: 10.1145/964723.383071. URL: https://doi.org/10.1145/964723.383071 (cit. on pp. 2, 6, 10, 16, 17, 19, 20, 24, 26, 34).

[41] 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 (cit. on p. 11).

[42] *The IPv4 Routed /24 Topology Dataset*. 2020. URL: https://www.caida.org/catalog/datasets/ipv4_routed_24_topology_dataset/ (visited on 11/30/2021) (cit. on p. 46).

[43] 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 (cit. on pp. 1, 9, 13, 14).

[44] J. Winick and S. Jamin. “Inet-3.0 : Internet Topology Generator”. In: *Distribution* (2002), pp. 1–19. URL: http://www.eecs.umich.edu/techreports/cse/02/CSE-TR-456-02.pdf (cit. on p. 46).

[45] W. Wu et al. “LDHT: Locality-aware distributed hash tables”. In: *2008 International Conference on Information Networking*. IEEE. 2008, pp. 1–5 (cit. on pp. 25, 33).

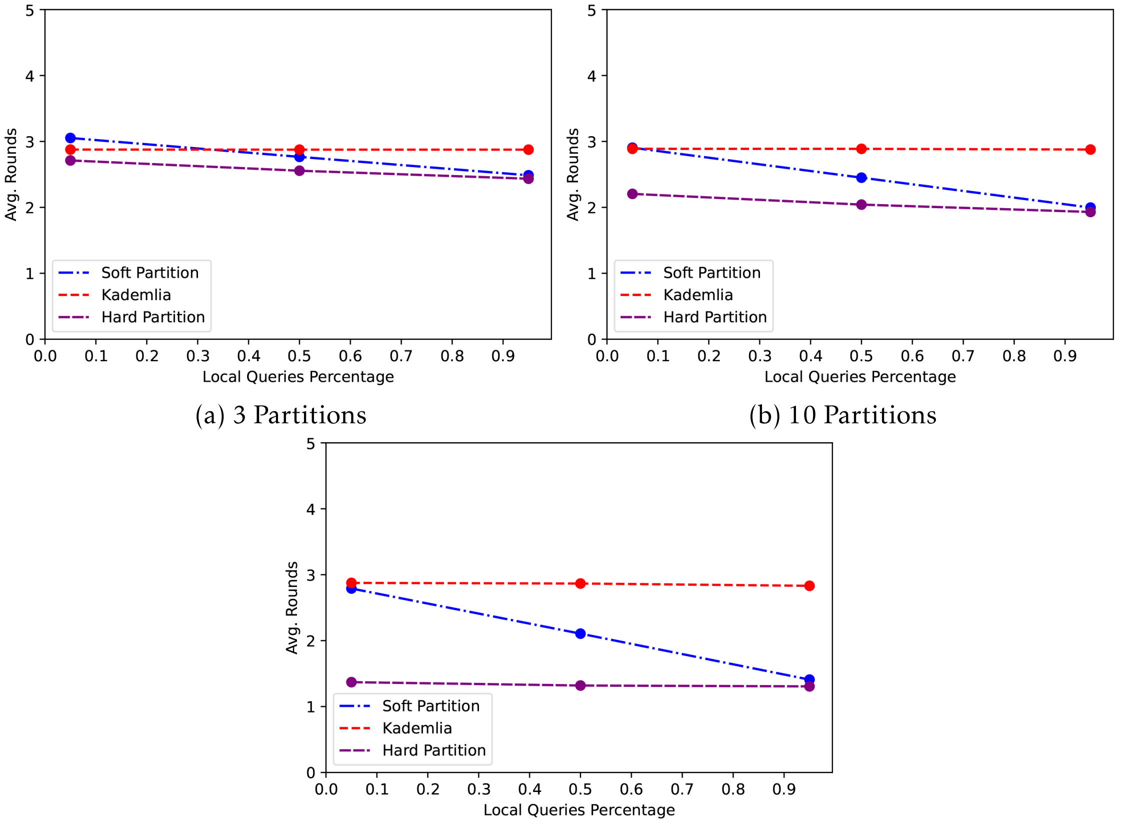
[46] B. Y. Zhao et al. “Tapestry: A resilient global-scale overlay for service deployment”. In: *IEEE Journal on Selected Areas in Communications* 22.1 (2004), pp. 41–53. ISSN: 07338716. DOI: 10.1109/JSAC.2003.818784 (cit. on p. 2).

[47] 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] (cit. on pp. 19, 22, 25).

I

Annex 1 - Extra Figures

This annex presents some plots that we did not include in the discussion in Section 4.3. This set was left out as we considered them to display similar results as the ones presented during the discussion. Nonetheless, we provide them here for completeness.



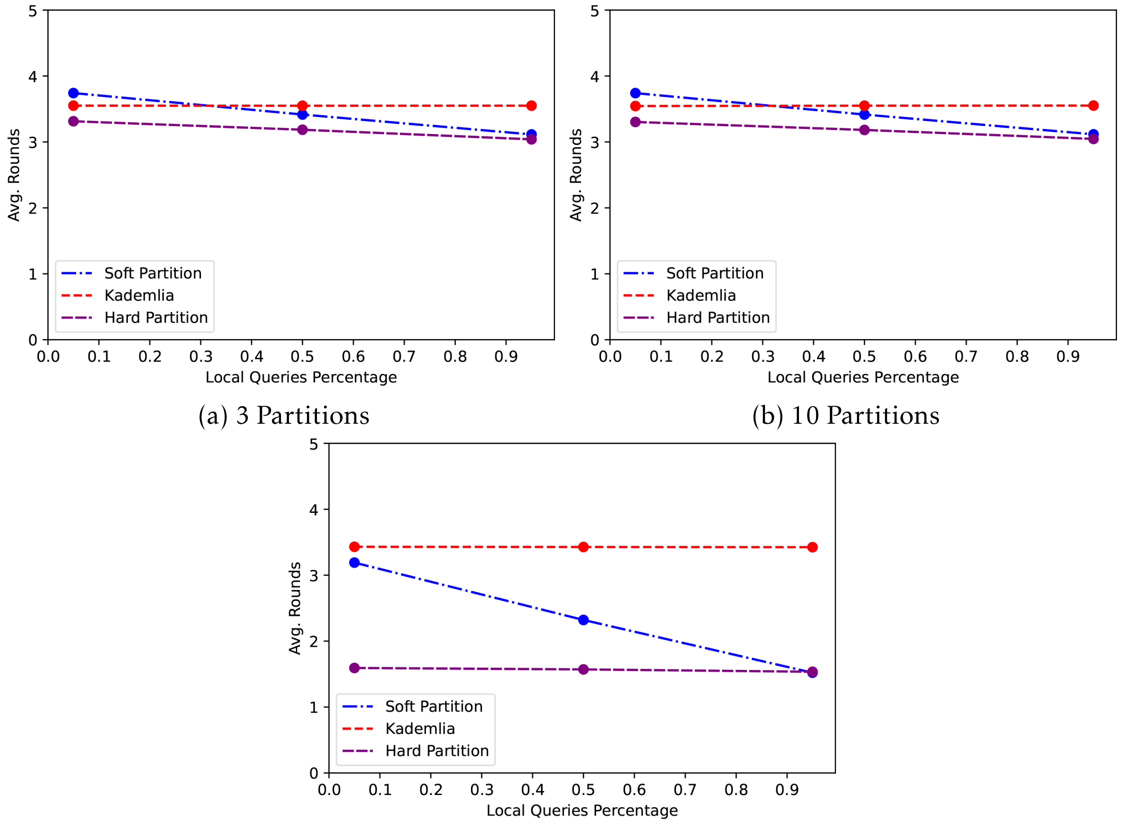
(a) 3 Partitions

(b) 10 Partitions

(c) 100 Partitions

Figure I.1: Average *FIND_NODE* average rounds with different partitions.

Figure I.1 shows the average Kademlia rounds for the same set of experiences as in Figure 4.5.



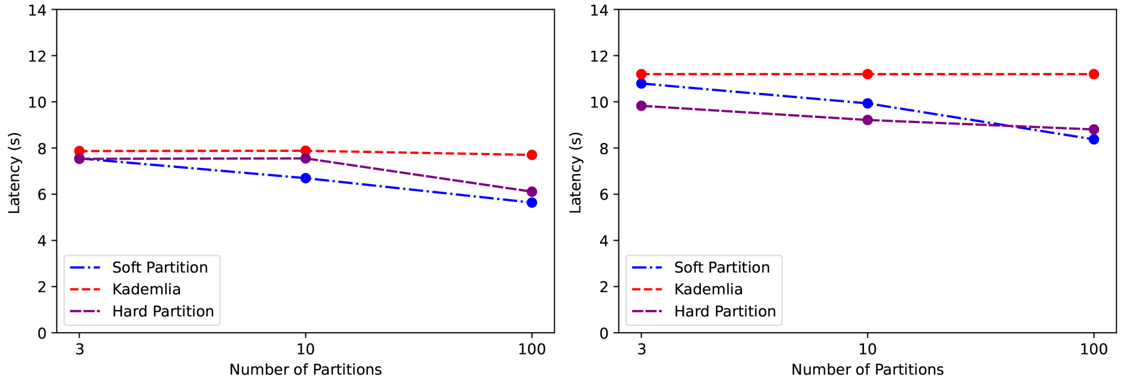
(a) 3 Partitions

(b) 10 Partitions

(c) 100 Partitions

Figure I.2: Average *FIND_VALUE* rounds with different partitions.

Figure I.2 shows the average Kademlia rounds for the same set of experiences as in Figure 4.6.

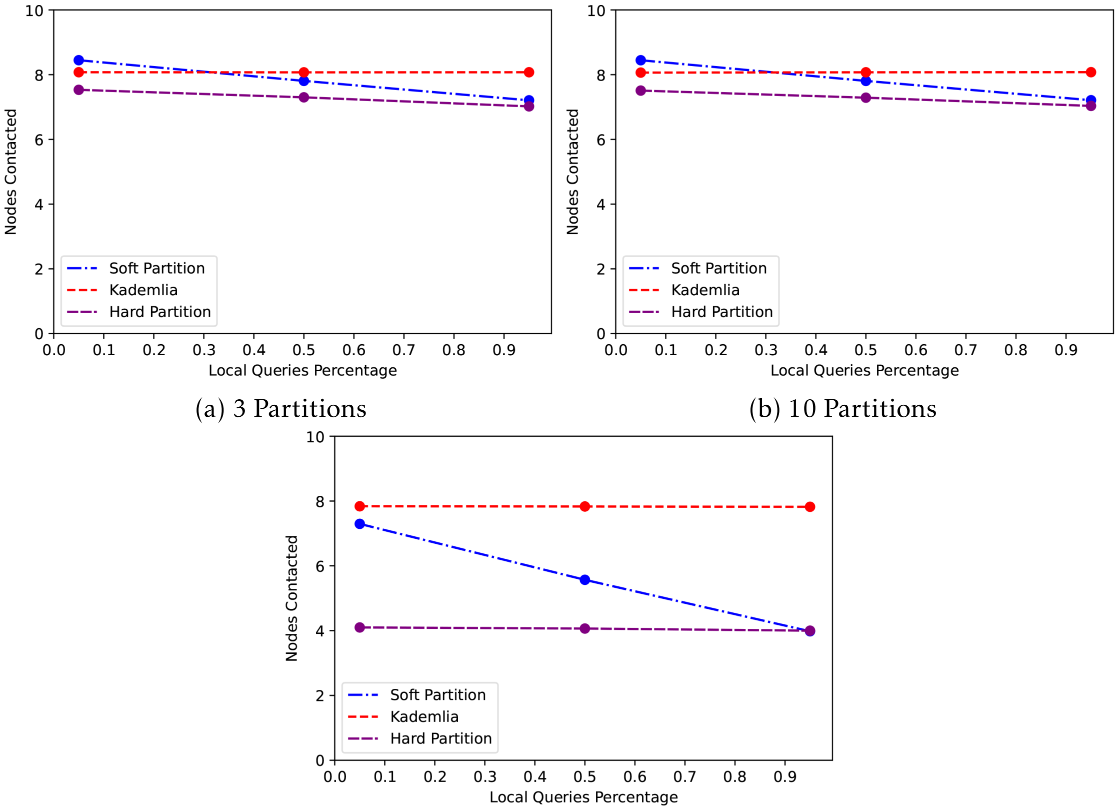


(a) Find Node

(b) Find Value

Figure I.3: Average query time with 50% bias.

Figure I.3 shows the average resolution time for both *FIND_NODE* and *FIND_VALUE* using a Biased Workload of 50%. The DHT parameters used are the same as in Section 4.3.3.



(a) 3 Partitions

(b) 10 Partitions

(c) 100 Partitions

Figure I.4: Average peers contacted in *FIND_VALUE* with different partition configurations.

Figure I.4 shows the average Kademlia rounds for the same set of experiences as in Figure 4.6.

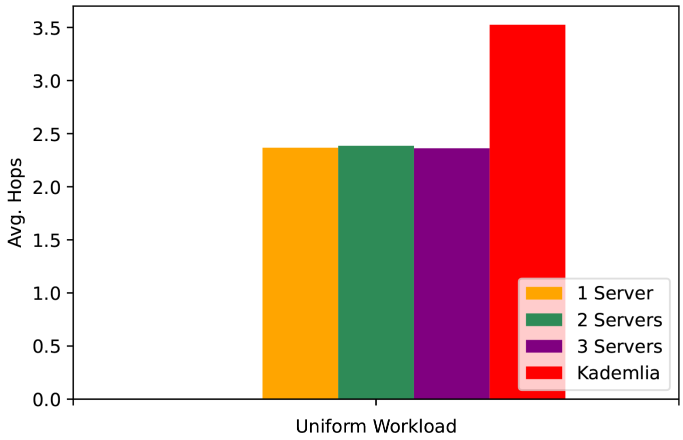
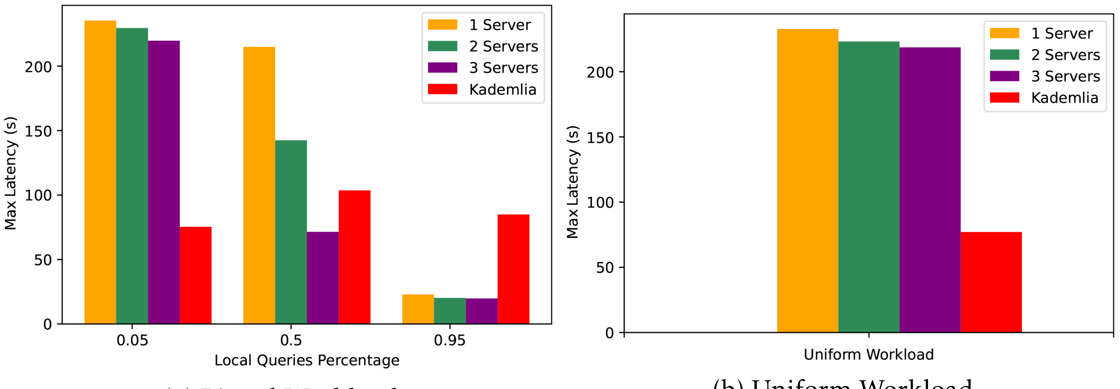


Figure I.5: Average *FIND_VALUE* query time to external partitions with multiple index configurations and a 100 partition configuration.

Figure I.5 shows the average hops for the Hard partition compared with Kademlia using a uniform workload, similarly to Section 4.3.3, under different index configurations.

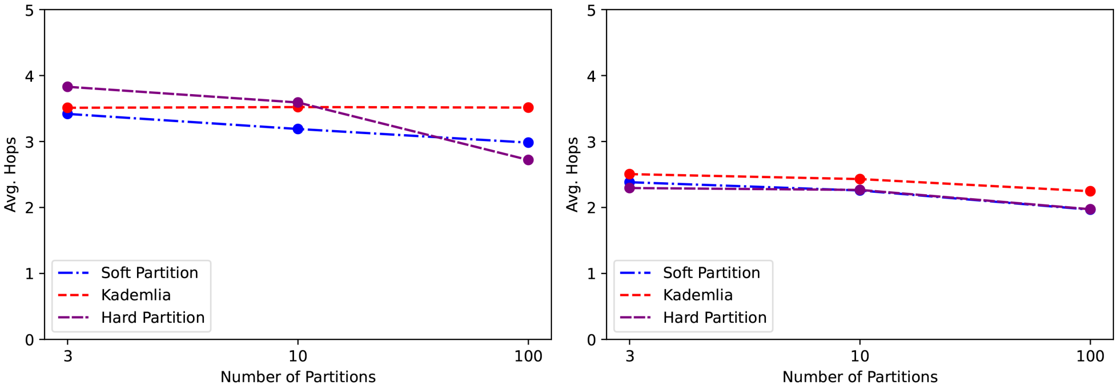


(a) Biased Workload

(b) Uniform Workload

Figure I.6: Maximum *FIND_VALUE* query time with multiple indexes in a 100 partitions configuration

Figure I.6 shows the maximum *FIND_VALUE* resolution time for the Hard partition compared with Kademlia using a uniform workload, similarly to Section 4.3.3, and a biased workload, similarly to Section 4.3.2 under different index configurations.

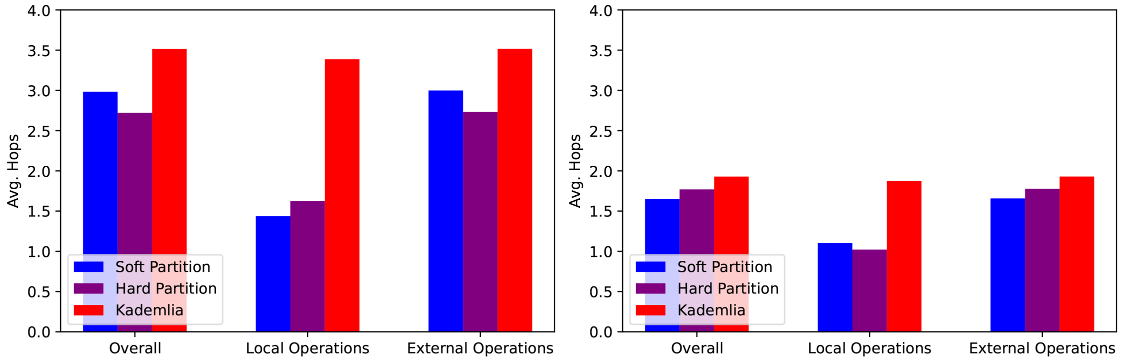


(a) Caching not enabled.

(b) Caching enabled.

Figure I.7: *FIND_VALUE* average hops with multiple partition configurations (Zipf workload).

Figure I.7 shows the average hops for the same set of experiences as in Figure 4.12.

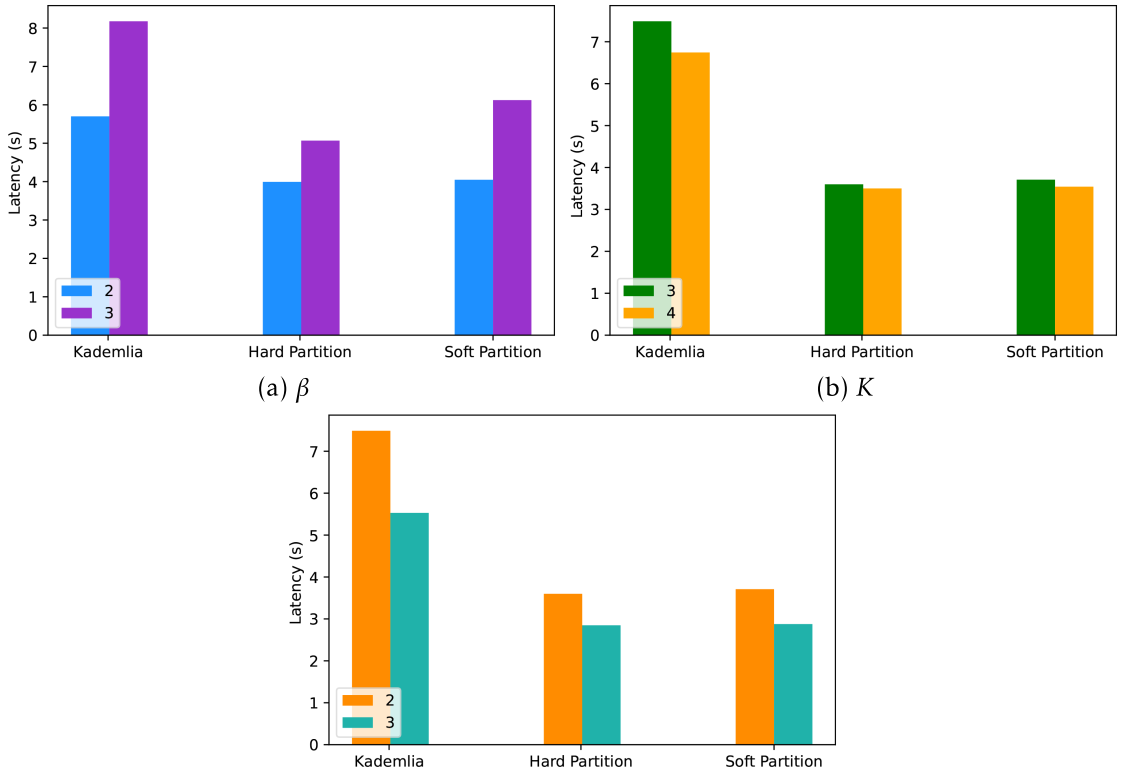


(a) Caching not enabled.

(b) Caching enabled.

Figure I.8: *FIND_VALUE* average hops in a 100 partitions configuration (ZipF).

Figure I.8 shows the average hops for a Zipf workload with a 100 partition configuration with the same DHT parameters used in Section 4.3.4.



(a) β

(b) K

(c) α

Figure I.9: Average *FIND_NODE* query time varying different Kademlia Parameters with 95% local queries.

Figure I.9 shows the average resolution time, using different Kademlia Parameters similarly to Section 4.3.5, with biased workload of 95% and a 100 partition configuration.

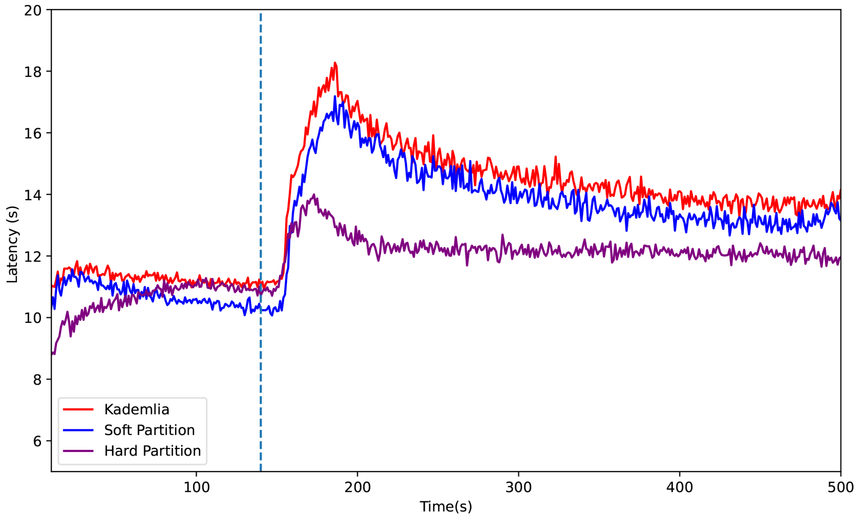
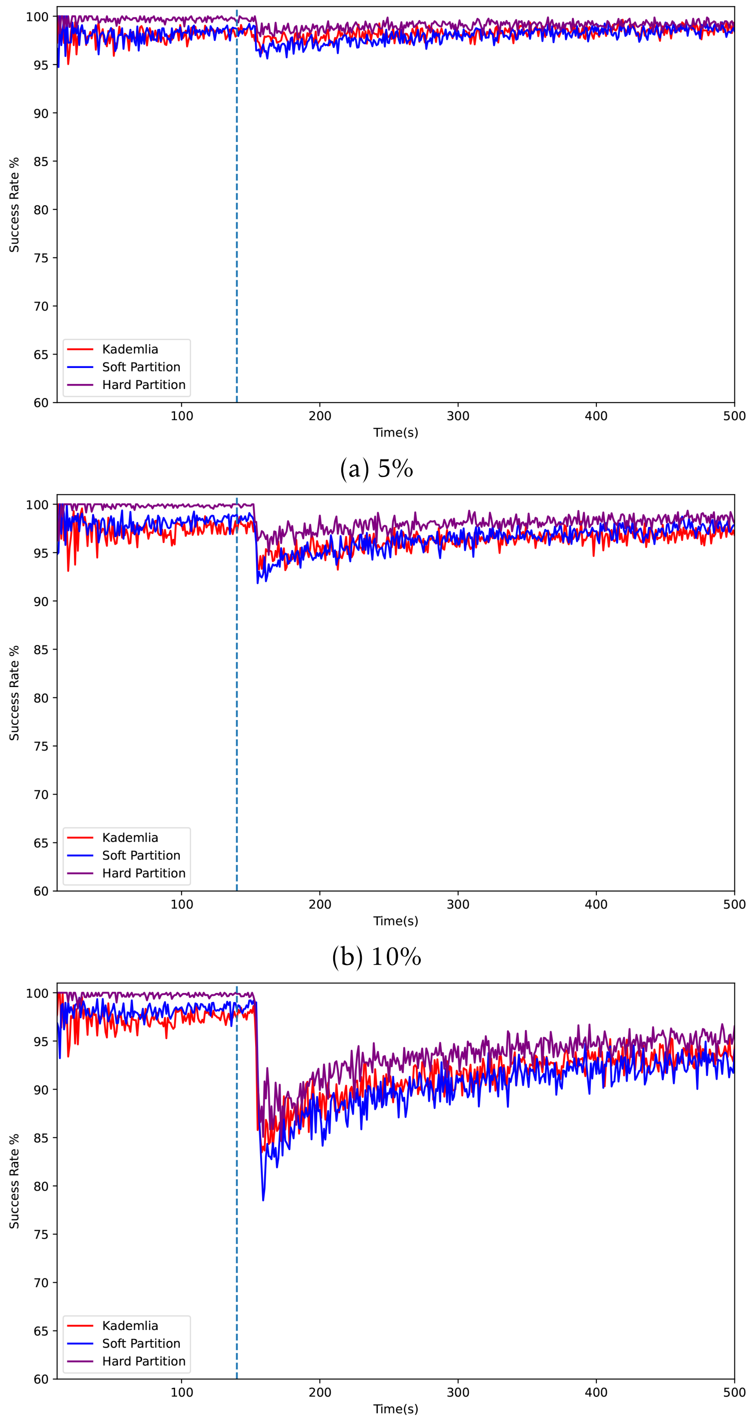


Figure I.10: Average *FIND_NODE* time with a 100 partition configuration and an instant failure of 30%

Figure I.10 shows the resolution time for the set of experiences presented in Figure 4.16b.



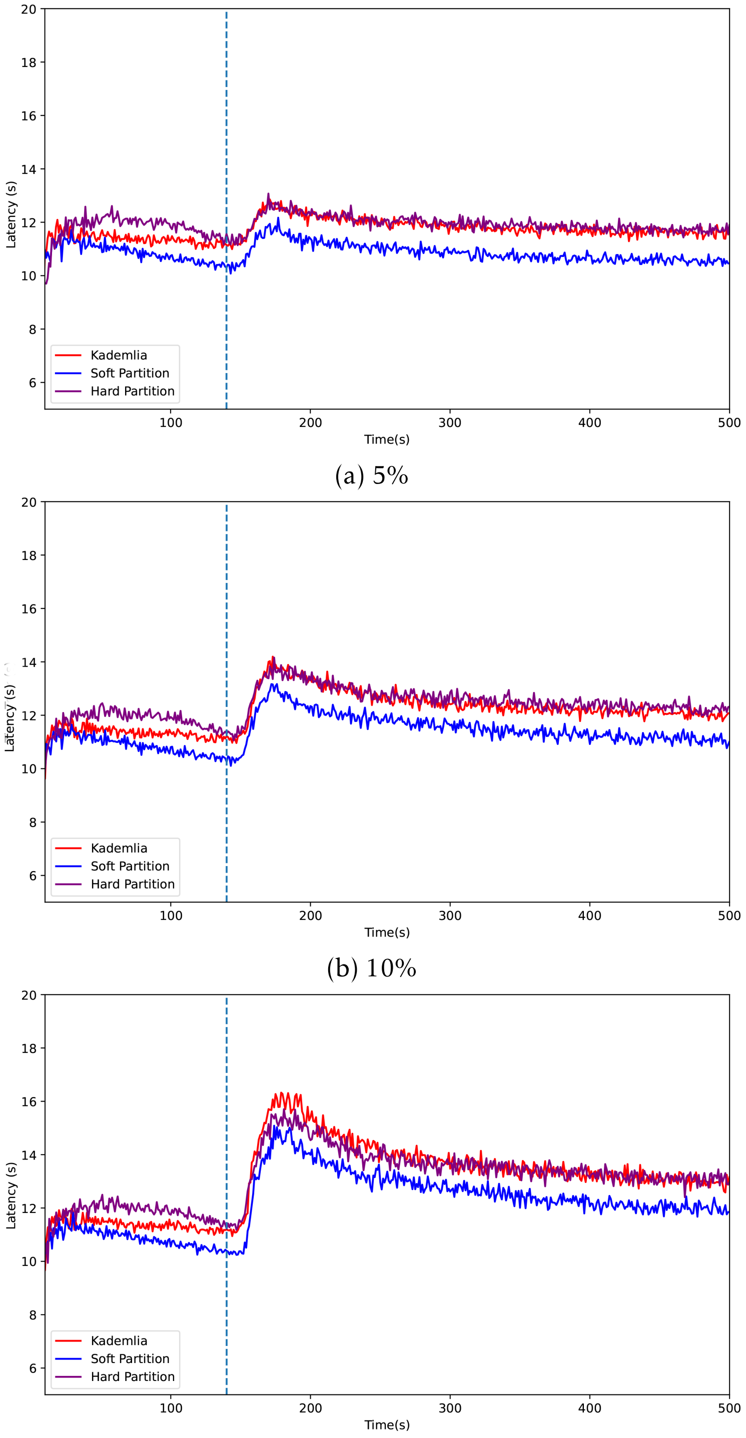
(a) 5%

(b) 10%

(c) 20%

Figure I.11: *FIND_NODE* success rate with different instant failure percentages and 10 partitions.

Figure I.11 shows the success rate for different failure rates using the experimental settings detailed in Section 4.3.6 and a 10 partition configuration.



(a) 5%

(b) 10%

(c) 20%

Figure I.12: *FIND_NODE* resolution time with different instant failure percentages and 10 partitions.

Figure I.12 shows the resolution time for different failure rates using the experimental settings detailed in Section 4.3.6 and a 10 partition configuration.