NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

TIAGO FILIPE VAZ ROCHETA DE MESQUITA GUERREIRO

BSc in Computer Science

EXPLOITING NODE CAPACITY IN DHTS: NODE AWARE LOAD BALANCING

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

February, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

EXPLOITING NODE CAPACITY IN DHTS: NODE AWARE LOAD BALANCING

TIAGO FILIPE VAZ ROCHETA DE MESQUITA GUERREIRO

BSc in Computer Science

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

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
February, 2022

Exploiting node capacity in DHTs: Node aware Load balancing

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.8.9) [1].

Abstract

Peer-to-peer systems overcome the limitations of centralized client-server model systems when it comes to scalability, fault tolerance and infrastructural costs. These systems grew in popularity over the years ever since services like Napster and Gnutella appeared.

However, what many P2P systems lack is the awareness of a individual node capacity when it comes to their load balancing strategies in structured overlays. Nodes are assumed to be equal in terms of their capacity in famous DHT protocols like Kademlia and Chord. In practical systems today, this is not true. Nodes can have different total bandwidth, processing power, storage latency and more. This *"blind"* approach to load balancing can lead to nodes potentially fail from overload and an overall decline of performance. Solutions for unstructured overlays exist but their translation to structured overlays is often not trivial.

In this thesis, we propose a variant of Kademlia that can be aware of node capacity, applies an appropriate load balancing strategy according to a node's capacity and is in the middle of the design space when it comes to structure.

Keywords: peer-to-peer systems, distributed hash tables, Load balancing in P2P systems

Resumo

Sistemas ponto-a-ponto superam as limitações de modelos de sistemas centralizados de client e servidor no que diz respeito à escalabilidade, tolerância a falhas e custos infraestruturais. Estes sistemas cresceram em popularidade ao longo dos anos desde que serviços como Napster e Gnutella apareceram.

No entanto, o que muitos sistemas P2P não têm é uma consciência da capacidade de um nó individual no que diz respeito às suas estratégias de equilíbrio de carga em sobreposições estruturadas. Os nós são assumidos como iguais em termos da sua capacidade em famosos protocolos DHT como Kademlia e Chord. Nos sistemas práticos de hoje, isto não é verdade. Os nós podem ter diferentes largura de banda total, poder de processamento, latência de armazenamento e muito mais. Esta abordagem *"cega"* para o equilíbrio da carga pode levar a nós potencialmente falhar devido à sobrecarga e à perda geral de desempenho. Existem soluções para sobreposições não estruturadas, mas a sua tradução para sobreposições estruturadas não é muitas vezes trivial.

Nesta tese, propomos uma variante de Kademlia que está ciente das capacidades dos nós, aplica uma estratégia de balanço de carga de acordo com a capaciade de um nó e situa-se no meio do espaço de desenho em termos de estrutura.

Palavras-chave: sistemas ponto-a-ponto, tabelas de dispersão distribuídas, balanço de carge em sistemas ponto-a-ponto

Contents

1 Introduction

1.1 Context

Peer-to-peer (P2P) systems have gained a lot of popularity over the years, providing scalability, fault tolerance, and reduced infrastructural costs that no centralized approach could. In these systems, typically peers (processes that belong and participate in system operations) are treated as equals with no hierarchy or bias, distributing the work among them all instead of delegating all relevant work to a single centralized entity. By sharing processing power, storage space, and bandwidth, all peers work together to provide a service. Since no individual peer is considered the *"server"* and all peers take on the roles of client and server simultaneously, having a single point of failure was never an issue for P2P systems. Infrastructural costs are reduced since peers need not be so powerful, and in turn expensive, due to shared resources they provide and receive usually to and from other peers.

For P2P systems, each peer usually only sees a partial view of the network and a distributed membership protocol is responsible for the maintenance of those views. This is essential to ensure scalability. With this, a logical network is created on top of the physical network called an *overlay*. Two main classes exist for overlays: *unstructured* and *structured*.

In unstructured overlays, creating links between peers is flexible as most of them are random in their nature [2]. This contributes towards a low maintenance overhead and in turn a more robust system even in highly dynamic environments. Unstructured overlays are usually used to support distributed systems with a gossip protocol, where nodes interact randomly to exchange information. These overlays are usually used for broadcast (the dissemination of messages to all nodes in a network) and the synchronization of replicas. Another use would be resource location, where peers discover and retrieve resources from one or multiple nodes responsible for the availability of those resources.

However, when it comes to structured overlays [2], a pre-specified topology must be

followed, creating constraints to membership protocols when creating links between peers. Usually, peers that join the system are attributed an *identifier* from an *identifier* space. The most popular usage of these overlays is to implement distributed hash tables (DHTs). DHTs provide application-level routing over the *identifier* space by using *keys* that belong in the same space as the node's *identifiers*. In these systems, messages are routed through the established links to reach nodes whose *identifier* is *"closest"* to the given key. Unfortunately, this types of overlay is much less robust compared to their counterpart due to the constraints in building links between peers and the higher cost of maintenance overhead to maintain the overlay topology.

Some overlays that belong in the structured category may in fact be considered a mix of both categories, exploiting benefits from both structured and unstructured overlays in attempt to reduce their drawbacks. A potential true *"hybrid"* of overlays will be proposed and discussed later in chapter 3, standing right in the middle of the design space between structured and unstructured.

1.2 Motivation

While the popularity of P2P systems as certainly escalated, progression and innovation did not follow the same path. Most moderns systems are based upon protocols with at least more than a decade old. One area which has been unexplored is the awareness of node capacity. One example of this can be found in the popular DHT protocol Kademlia [3].

Kademlia is a structured overlay, more specifically a DHT, that provides a low maintenance overhead system with a closest k nodes topology. Kademlia's topology can be considered very relaxed, since nodes can be added to each others state with much less restraint. Maintenance is mainly performed during normal lookup requests instead of dedicated procedures and messages like other overlays have. Lookups tend to route to the same path regardless of the starting node, passing through the same nodes. Like other DHTs, Kademlia also distributes content evenly among nodes, due to the random distribution of *identifiers*. This shows that Kademlia (and others) assume that nodes are created equal when it comes to their capacity. In practice, in many practical systems; this is not the case. Nodes have different processing power, storage latency and even available bandwidth in modern systems.

In this thesis, we will explore the design space of DHTs, starting with the design of Kademlia, by adding constraints towards linking peers together. These constraints will be based on the node's current capacity and will change throughout the node's stay in the system. The more capacity (e.g.: more processing power, lower storage latency and higher bandwidth) a node has, the more other nodes should know the node's existence, increasing the amount of queries routed to the node. The less capacity a node has, the less other nodes should know of the node's existence, to avoid overloading.

1.3 Expected Contributions

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

- The implementation and evaluation of an hybrid overlay, that further enhances the original design of DHTs with fine tuned constraints based on node capacity.
- An experimental comparison between the aforementioned design and the original implementation of Kademlia and other DHTs.

1.3.1 Research Context

The work to be performed during this thesis is conducted by NOVA School of Science and Technology in association with Protocol Labs. Protocol Labs is a company that specializes in the development of P2P systems, protocols, services and frameworks. Popular examples of their work include libp2p [4], Filecoin [5], and IPFS [6].

Both libp2p and IPFS are Kademlia based, which inspired us to design space around a protocol like Kademlia that is self-aware of the heterogeneity of capacity among participating nodes.

1.4 Document structure

The rest of the document is structured as follows:

- In Chapter 2 we will talk about P2P systems, their applications, services and overlay networks. In further sections, we will also discuss types and metrics of overlay networks, focusing more on examples of Distributed Hash Table protocols like Chord and Kademlia. However, some unstructured overlays will also be discussed briefly to contribute for more context, and because some of the techniques employed in the design of those solutions could benefit us. Furthermore, we will discuss load balancing in P2P systems, focusing on the lack of solutions for structured overlays.
- In Chapter 3 we will elaborate our proposed solution, tasks to be performed, evaluation of our solution and the work schedule.

2

Related Work

In this chapter, we will first introduce and study relevant topics and concepts that are the base of the elaboration of this thesis while also presenting related work. The chapter structure is as follows: in Section 2.1 we will introduce what are peer-to-peer systems; in Section 2.1.1 we will present some uses cases for P2P systems; in Section 2.2 we will introduce what is an Overlay network; in Section 2.2.1 we will briefly explore unstructured overlays and some of their examples; in Section 2.2.2 we will introduce what is a structured overlay and briefly describe their most common type; in Section 2.3 we will explore what is a DHT and describe in detail some of their famous examples; in Section 2.4 we will discuss some solutions to load balancing in p2p systems and the lack thereof for structured overlays; and finally, in Section 2.5 we will reflect on the most observations to justify our solution.

2.1 Peer-to-Peer Systems

A Peer-to-Peer (P2P) system consists of a network of multiple nodes usually without hierarchy, running software with similar functionality [7] and sharing computational power, storage, and/or network bandwidth to other nodes to work towards a common goal. This is a different approach compared to the typical client-server model which has its own limitations including in form of scalability, fault tolerance, and operational costs. P2P systems overcome these limitations of the traditional client-server model by being decentralized, with no one node being more essential than the other.

Scalability is addressed via sharing of each participant's own computational power, storage or network bandwidth to other participating nodes. Fault tolerance is increased by P2P networks lack a single point of failure. And, finally, operational costs are reduced by avoiding powerful and expensive server that is capable of handling a large quantity of

clients at the same time [2].

P2P Architecture

A P2P system has 4 essential layers in its architecture: application, service, overlay network and physical network. The order of dependency goes from top to bottom (top depends on bottom) as seen in Figure 2.1. In this work, the focus will be in the upper layers, specifically Overlay Network layer, touching upon the application and service layer since they depend on the overlay network below them.

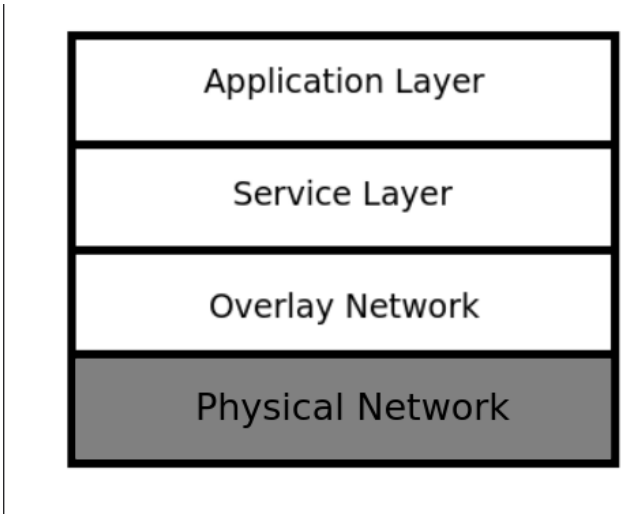

Figure 2.1: P2P Architecture adapted from [2].

Physical Network

This layer receives, sends and queues messages from (or to) other peers using the transport layer and acts as an interface to that layer for the above layer. Most of the time, this is masked by the IP network and a set of relevant transport protocols, including TCP, UDP, and more recently QUIC [8].

2.1.1 P2P Applications and Services

One of the most popular use cases for P2P systems is resource location, which was popularized by applications like Napster [9] and Gnutella [10]. Participants would obtain identifiers of a set of peers that own a given resource. The resource itself does not need to be specified, it could be a file, free CPU time, an entry to a distributed database, or other [2].

Another common example of a service would be application-level routing solutions. The service would provide efficient and up-to-date route paths for messages to be sent through to other peers, enabling point-to-point communication at the application layer.

Lastly, P2P can be used to implement publish and subscribe services. These types of services serve so that users can create topics and subscribe to them. The owner of a topic can publish new entries and they will be distributed by dissemination to all nodes that have subscribe to said topic.

2.2 Overlay Networks

The overlay network layer manages the logical network that contains all participating peers in the system. Each peer runs a distributed membership protocol with a set of contact information of peers (usually IP address and Port) that are called neighbors, linking peers together to create the logical network. This set changes as peers leave and join the network which is handled by the membership protocol, notifying the above layer of these changes. P2P overlays consist of two classes depending on how the overlay is maintained and created. In this paper we will focus more on structured overlays, specifically Distributed Hash Tables (DHTs).

2.2.1 Unstructured Overlays

In this type of overlay, nodes are organized in a random topology, with each node creating arbitrary connections between other nodes. Because of this, these types of networks have a high fault tolerance but become less efficient when locating a specific resource in the network. These types of overlays are usually used to support range and keyword queries in resource location services. Usually nodes in these overlays keep *partial views*, a set of nodes that represent a literal partial view of the network in the perspective of the current node. Queries to locate resources are disseminated through out the whole network to guarantee that all possible values are returned. Since our focus is mostly about structured overlays, specifically DHTs, we will briefly explain some examples of unstructured overlays.

2.2.1.1 HyParView

HyParView [11] protocol is know for having two partial views, with distinct purposes, properties, and maintenance strategies.

active view A small set of nodes that each participating node has. A TCP connection is kept with each node in the set. All links in this view are symmetric, if node n has node p in its active view, node p must also have node n in its own active view. This view is used mainly for message dissemination and detecting node failures at each broadcast.

passive view A larger set of nodes (at least k times larger than the active view) that each node uses as a quick source of replacements in case a node in the active view fails.

This view is maintained with periodic shuffle procedures that exchange known contacts to the nodes in the active view.

In the shuffle procedure, not only nodes from the passive view are exchanged but also nodes from the current node's active view. This increases the probability of having active nodes in passive views and evicts failed ones eventually. HyParView can remove failed nodes and replace them fairly quickly due to using TCP connections on nodes in the *active view* and actively maintaining a large set of replacement nodes in the *passive view*.

2.2.1.2 Cyclon

In Cyclon [12], each peer maintains a set of nodes (more precisely, information about those nodes) that are called *neighbors* and perform periodically but not synchronized simple shuffle procedure to exchange *neighbors*. Each *neighbor* has an *age* field that helps to estimate when was the last time the current node's could confirm a certain *neighbor* was alive. This field is relevant for Cyclon's shuffle procedure, performed by each node periodically.

The shuffle procedure starts by increases the *age* field of every *neighbor*. Next, it picks the eldest neighbor as its target and *l-1* random other *neighbors* to the set *S*. The current node swaps the targets entry in the *s* set for its own entry with *age* 0 and sends the modified set to the target. The target replies with a random set of at most *l* of its own neighbors. Cyclon can be very robust since even when half of the nodes fail simultaneous, connectivity is not threatened.

2.2.1.3 SCAMP

SCAMP [13] nodes keep a *partial view* that scales its size with the system size, even though no individual node knows the system size. This is due to the way SCAMP handles new nodes to join the network. When a node *n* tries to join, it sends a subscription request to the contact node *p* and disseminates a forwarded subscription request to all nodes of its view plus *c* copies of the forwarded request to randomly selected nodes in the view. Each node receiving a forwarded subscription request have chance to not add the new node to their view and forwarding the request to a random node in their view. This probability gets higher as the view size grows, eventually growing he view size based on how many nodes exist in the network.

2.2.1.4 Gia

The Gia [14] protocol accepts and uses the heterogeneity of nodes in terms of their capacity to achieve better scaling. Nodes with high-capacity in a Gia network are ensured to receive more queries and be more connected (e.g.:a higher degree) than low-capacity ones. Gia also actively tries to avoid hotspots with flow-control tokens given to nodes based on

available capacity. Moreover, Gia nodes offers one-hop replication of pointers to content. Nodes maintain pointers of the content provided by their immediate neighbors, ensuring high-capacity and degree nodes are capable of providing answers to a large amount of queries. And finally, Gia relies on biased random walks that tend to direct queries to high capacity nodes, which usually are best to answer queries.

2.2.2 Structured Overlays

Unlike unstructured overlays, structured overlays organize themselves into a specific topology, imposing constraints when creating links between nodes. These constraints help locating resources more efficiently but also create maintenance overhead, since peers are not as *"free"* to join or leave as they are in unstructured overlays. In these overlays, nodes usually are assigned *identifiers* that (at least partially) help decide when creating links between nodes.

The most common type are DHTs, where consistent hashing is used to distribute and assign ownership and responsibility to peers for storing a file. Usually, files are paired with keys that are used to query into finding the peer responsible for that specific file and in this paper we will talk about keys as the *"identifier"* of a certain file in a DHT and sometimes even referring to the key as the file itself.

In this paper, we will focus more on DHTs and explain a few examples, describing their lookups, maintenance, strengths, drawbacks and even some of their variants and what they do the enhance the original. Since our work will be mostly focused on DHTs, we discuss this type of overlay at length in Section 2.3.

2.2.3 Relevant Metrics

Metrics are used to evaluate performance of an overlay network. This is useful to evaluate what trade-offs are made, what situations do overlays perform better or worse, or to decide best use cases for them. Some of these metrics [15] that we will explain are: Degree, Hop count, Degree of fault tolerance, Maintenance overhead and load balance.

Degree Distribution

The degree of a node is the number of neighbors that each node must keep in contact. This metric is useful to determine robustness of an overlay since it exposes weakly connected and massively connected nodes [12]. These type of nodes not only show the unfair distribution of resource usage like processing and bandwidth but also how if a few certain nodes happen to fail, a big chunk of the network may become isolated, reducing if not ruining message routing and by consequence lowering performance. The degree of a node can be divided in to types: *in-degree* and *out-degree*.

in-degree of node n refers to the number of other nodes that know n's contact information. Node n does not need to know any of these nodes' contact information. This number helps estimate the quantity and likelihood of queries will reach n. The higher the in-degree of n, the more likely queries will pass through and reach n.

out-degree of node n refers to the number of nodes' contact information n possesses. Similarly, these nodes do not need to know n's contact information. This number affects n's routing decisions. The higher the out-degree of n, the more likely n's routing decision will be better.

Hop count

The number of intermediate nodes a message passes through from any source node to any destination node. This metric establishes a notion of how many nodes does on message have to visit in order to reach its destination. The metric by itself doesn't provide much since it depends on the size of the network. However, the lower the hop count, the less nodes a message has to go through, the lower the latency and the less likely the message will be lost and not reach its destination.

Factor of fault tolerance

The percentage of nodes that can fail without losing data or preventing successful message routing [15]. This metric is important to ensure the practicality of the overlay, since all nodes will eventually fail unexpectedly and sometimes even simultaneously. A low degree of fault tolerance is undesirable for any overlay. The higher the degree is the more nodes can fail without affecting data integrity and correct message routing. In unstructured overlays, this degree tends to be higher than structured ones due to the fact that for the same number of nodes failing it is irrelevant which exact nodes fail in an unstructured network. In a structured one, certain combination of nodes may lead to disruption of the established structure which may result in lost of performance.

Maintenance overhead

How often messages pass through nodes and their neighbors to maintain coherence as new nodes join and nodes leave or fail [15]. This is important to establish how much background work will nodes do and how much processing power is used to perform this work. High maintenance overhead may expose lack of performance in overlays if they do not provide a reasonable trade-off in their guarantees or other metrics. This metric is usually high in structured overlays than unstructured ones due to the fact that messages are used to replace failed nodes and integrate new ones. For new nodes to be integrated, special rules and procedures are applied in order to maintain a specific structure in structured networks.

Load balance

How much load does each node experience as an intermediate node and how even the keys are distributed across nodes [15]. This helps to assess resource usage distribution across nodes. Typically, a *"fair"* load balance would be an the total spread evenly across every node, independent of a node's capacity. However, in practical systems, nodes have different capacity. Since capacity can be different among nodes, distributing load evenly without consideration for the node's individual capacity may not be as fair as one might assume. As we will see in Section 2.3, many examples of structured overlays (in this specific case DHTs) assume all nodes to have the same capacity. We will discuss this topic in more detail in Section 2.4.

Degree of structure in Structured Overlays

Usual performance metrics for structured overlays focus on hop counts, fault tolerance, load balance,degree and maintenance overhead [15] but we observed that there might be a unexplored metric: spectrum of structure of the resulting logical network. The drawbacks and advantages of having a particular topology may affect other metrics more than we expect. We will detail this topic (specifically for DHTs) further in Section 2.3.6

Next we will explain what is a DHT and provide some examples of DHTs including a discussion on their strengths, their drawbacks, and what trade-offs do their variants provide. When referring to a key k that is used to refer to some content that may or not be stored, the ID of the referred key is also k and when referring to a node n, the ID of the referred node is also n.

2.3 Distributed Hash Table (DHT)

A DHT is essentially a hash table spread across multiple nodes, used for lookup just like a normal hash table. The goal of a DHT is to provide lookup for content, either be a simple value, document, image, video or any kind of arbitrary data, independent of where that content is exactly stored with reasonable response time [7]. DHTs can be implemented with a strong sense of structure like Chord, where all nodes have *identifiers* that can be organized in a *identifier* circle(or ring) or much less structured like Kelips' affinity groups, contacts, and filetuples.

2.3.1 CAN

CAN [16] nodes *"owns"* a partition of a virtual d-dimensional Cartesian coordinate space on a *d-Torus* that has no relation to any physical Cartesian coordinate system. An example of this space, although simplified to be easier to understand, see Figure 2.2. The entire coordinate system is dynamically partitioned among all nodes so that nodes own a zone, distinct within overall space. Storing (key,value) pairs in this coordinate spaces is as

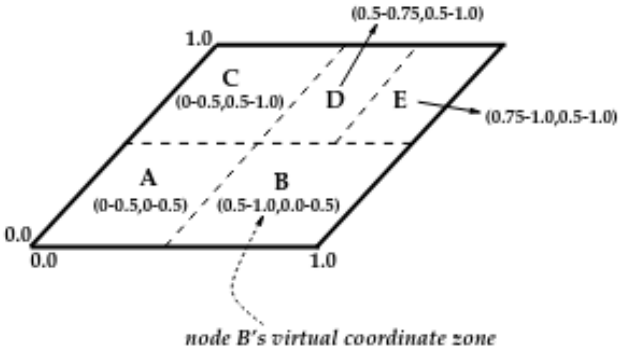

Figure 2.2: Illustration from [16]. An example 2D space with 5 nodes. Note: this is a simplified example, the reader must remember that the coordinate space wraps.

follows: to store a (key,value) pair with key K, a deterministic hash function is used to deterministically map onto a point P in the coordinate space. The node that ones the zone where P resides, is responsible for storing the (key,value) pair with key K.

From its inception, CAN protocol was not limited to being used in P2P systems. It can be used in large scale storage management systems like OceanStore, Farsite, and Publius. As of today, CAN is less and less being used as a DHT, and for that we will not discuss its variants.

Node state and maintenance

CAN nodes keep the IP address of the owners of adjacent zones to their own zone as neighbors. When a node wants to retrieve an entry, it applies the same deterministic hash function to map key K onto point P. If P does not belong to the current node or its immediate neighbors, the request is routed in the CAN infrastructure until it reaches the node that owns the zone where P lies.

When a new node n joins a CAN network, in order to assign a new zone to n, an already owned zone is split into two, one half for the original owner and the other for n. CAN does not depend on the details on how its done but the authors used the same bootstrap mechanism as YOID [17]. CAN assumes it has a DNS domain name that resolves to the IP address of one or more CAN bootstrap nodes. A bootstrap node maintains a partial list of all nodes that it believes are in the system. Node n would look up the CAN domain name in the DNS and get a bootstrap node's IP address. Then, the bootstrap node provides several IP address of randomly chosen nodes in the system.

When assigning a new zone to n, n randomly picks a point P in the space and sends a join request with P as its destination. After this request is routed to the node that owns the zone where P lies, that node splits the zone in half and assigns one half to n. This split is done in way that makes it possible for the zones to be re-merged when nodes

leave. After the split, the (key,value) pairs of the half and the appropriate neighbors get transferred to node n.

Every node sends an immediate update message, followed by periodic refreshes, with its currently assigned zone and all of its neighbors to update affected neighborhoods by the new node joining. This ensures all neighbors of the n and the node that split the zone will learn about the change and their neighbor set.

Since nodes send periodic update messages, the prolonged absence of these messages indicates node failure. When a node notices this absence and decides the node has died it initiates the takeover mechanism and starts a takeover timer. Each neighbor of the failed will do this independently, with the timer initialized in proportion to the volume of node's own zone. When the timer expires, the node sends a TAKEOVER message to all neighbors of the failed node with its own zone volume included in the message. Once a node receives a TAKEOVER message, the node cancels its TAKEOVER timer if the zone volume in the message is smaller than its own. Otherwise, it replies with its own TAKEOVER message. This effectively ensures the chosen node to take over the zone is alive and has the smallest zone volume.

In case of simultaneous node failures occur, the node that detects these failures performs an expanding ring search for any nodes residing beyond the failure region before starting the takeover. This way the node eventually gains sufficient neighbors to initiate takeover safely.

Finally, the normal leaving procedure and the immediate takeover algorithm may result in a node owning more than one zone. CAN runs a background zone-.reassignment algorithm to ensure nodes have only one zone.

Lookup

CAN's lookup can be compared to following a straight line through a Cartesian space from starting coordinates to end coordinates. Two nodes are neighbors in a d-dimensional space if their coordinates spans overlap along $d - 1$ dimensions and adjacent along one dimension. In simpler terms, two nodes are neighbors if all but one dimension share the same overlap of values and the one dimension they do not share is adjacent to one another. As we can see in the Figure 2.3, node 1 is a neighbor of node 7 because node 1's coordinate zone overlaps with 7's in the Y axis but is adjacent along the X axis. Node 6 is not a neighbor of node 1 because both X and Y axes of 6 are adjacent to 1's axes. This neighbor state is enough to route between two arbitrary points in space. A CAN message contains destination coordinates and using a simple *greedy* algorithm, CAN forwards messages to the neighbor with the closest coordinates to the destination coordinates.

Even when nodes fail, alternate (most likely longer) paths can still be used for routing. If for some reason a node becomes isolated with no neighbors in their neighbor set, the *greedy* forwarding of messages may fail. To avoid this, CAN uses an expanding ring

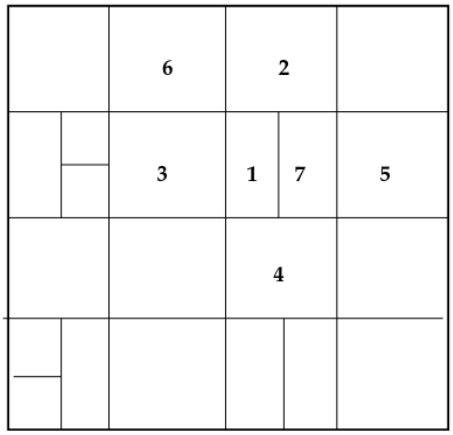

Figure 2.3: Illustration from [16]. Example 2D space with some numbered nodes and their zones.

search (essentially a stateless and controlled flooding) to locate the node that is closer to the destination than the current node. When that node is found, forwarding of the message can resume.

Strengths and drawbacks

CAN's most obvious strength is the lightweight node state since it scales with dimensions more than network size. It is also a protocol that can be implemented entirely on the application level.

However, CAN's coordinate space partitioning can hinder not only performance but also potential loss of content when faced with many nodes failing simultaneously that own adjacent zones.

2.3.2 Pastry

Pastry [18] nodes are assigned random 128-bit IDs which indicate the node's position in a circular ID space, which ranges from 0 to $2^{128} - 1$. In a network consisting of N nodes, Pastry route messages to the closest node to a given key in less than $\lceil \log_{2^b} N \rceil$ steps(b is a Pastry configuration parameter with a typical value of 4). Despite node failures, Pastry ensures eventual delivery if $\lceil |L|/2 \rceil$ nodes with *adjacent* IDs did not fail simultaneously ($|L|$ is a configuration parameter with a typical value of 16 or 32). For routing purposes, IDs and keys are thought of as sequences of digits with base 2^b. When forwarding messages, Pastry chooses the node whose ID a prefix with the key that is at least one or b bits longer than the current node's prefix shared with the key. If the current node does not know of such node, Pastry picks a node with the same prefix shared length but numerically closer to the key than the current node ID.

Node state and Maintenance

Each node stores and maintains a routing table R, neighborhood set M and a leaf set L. The routing table has $\lceil \log_{2^b} N \rceil$ rows with $2^b - 1$ entries each. An entry at row n is an ID of a node that shares the first n digits with the present node's ID, but the $n + 1$th digit is different value from $2^b - 1$ total possible values. As we can observe in the Figure 2.4, the node 10233102 has their routing table filled with entries of node IDs, with each row having a shaded cell indicating the current digit of the present node's ID and each entry node ID being structured as such: prefix digits that are shared with the present node's ID - different digit - rest of the ID. Each entry consists of an IP address of one of the potentially many nodes that have the appropriate prefix. In practice, nodes are chosen according to a proximity metric which we will discuss better later in this section when explaining maintenance.

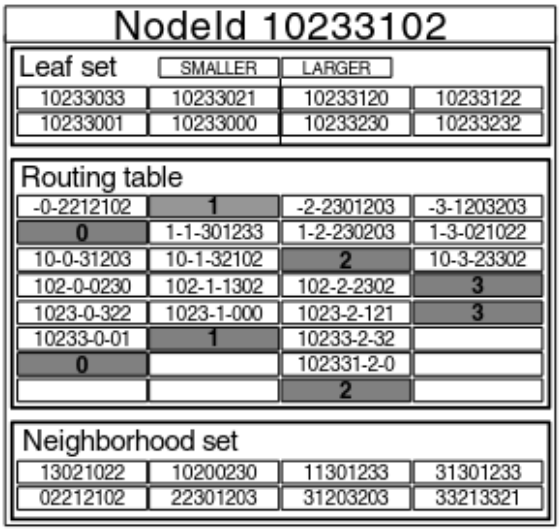

Figure 2.4: Illustration from [18]. State of a node whose ID is 10233102. $b = 2$ and $l = 8$. All numbers are in base 4. Node IDs are 16 bits long. IP addresses are not shown.

The parameter b involves a trade off between the size of the maximum number of useful entries (approximately $\lceil \log_{2^b} N \rceil \times (2^b - 1)$ entries) and the maximum number of required hops between any two nodes ($\lceil \log_{2^b} N \rceil$).

The neighborhood set M contains node IDs and IP addresses of $|M|$ nodes that are closest to the local node, according to the proximity metric. The leaf set L is the set of nodes where half of the nodes have numerically closest larger IDs, and the other half of the nodes have numerically closest smaller IDs, relative to the current node's ID. Typical values for the size of both of these sets are 2^b or 2^{b+1} each set.

Pastry's notion of network proximity is based on a scalar proximity metric, for example geographic distance. Pastry nodes assume the above application provides a function to *"determine"* distance of a node with a given IP to itself. Nodes with a lower distance value

are considered more desirable. The entries in the routing table of each node are chosen to be close to the node, according to the proximity metric, among all appropriate nodes with the desired ID prefix. This metric is used not only to fill the neighborhood set but also to fill the routing table with the topologically closest and appropriate node for a desired prefix.

When a new node X arrives, Pastry assumes the node knows about a nearby Pastry node A, according to the proximity metric, that is already part of the system. After initializing its state tables, node X asks node A to route a special *join* message with the key being equal to X. Like any usual message, A routes the message to a node whose ID is numerically closest to X, which we will refer to as node Z. Upon receiving the special message, nodes A, Z and every node that the message has passed through to reach Z will send their state table to X. Since A is closest node to X according to the proximity metric, A's neighborhood set will serve to initialize X's. For the leaf set, node Z has the closest ID to X, thus its leaf set will serve as basis for X's leaf set. As for the routing table, assuming the most general case that A and X do not share a common prefix in their IDs, node A's row 0 can serve as X's row 0, since row 0 is independent of the node's ID. The rest of the rows of A serve no purpose since A and X do not share a common prefix.

However, node B's, where B is the first node the message passed through between A and Z, row 1 are appropriate values for row 1 of X. And in a similar fashion, node C's row 2 are appropriate values for row 2 of X, node C being the second node encountered between A and Z, and so on. Finally, node X sends a copy of the resulting state to each of the nodes found in its routing table, leaf set and neighborhood set. Those nodes in turn update their own state based on the information received from X. When sending state information, nodes attach a timestamp on the initial message and the receiver node replies to notify the other node of the message's arrival, with the original timestamp attached so that if state has changed since the timestamp, an update message is sent.

Lookup

In Pastry, the main procedure is used to determine what is the next node to forward a certain message. Assuming a message with key D arrives at node with ID A, the procedure is as follows. If the key D does fall within the range of IDs covered by the leaf set, the message is forwarded directly to the destination node, namely the node in the leaf set whose ID is closest to the key D. Otherwise the routing table is used and the message is forwarded to the node that shares a common prefix with key D by at least one more digit than node A. In certain cases, such node may not exist in the routing table and the message will be forwarded to the node with a shared prefix with the key D at least as long as node A, and is numerically closer to the key than A. Such a node must exist in the leaf set or the message has already arrived to the numerically closest node ID. Unless $\lfloor |L|/2 \rfloor$ adjacent nodes have failed simultaneously, at least one of those nodes must be alive.

Strengths and Drawbacks

Pastry's main strength relies on not only the efficient routing of multiple hops but also efficient routing of a single hop. Because of its locality, nodes in the routing table in Pastry are topologically closer, and therefore potentially provide less latency when routing. This gives an edge on latency for Pastry compared to other overlays.

Unfortunately, Pastry's maintenance is complex and costly for highly dynamic systems. If a node experiences a problem, Pastry will perform costly procedures in order to remove that node from state and replace and reorganize its node's states.

Variants

Scribe

Scribe [19] itself is not a variant of Pastry, but a publish/subscribe service built on top of Pastry. With a publish/subscribe model, users can create topics, subscribe to them and publish events on topics so that those can be disseminated to all of the topics subscribers. Scribe uses pastry for topic creation, subscription and to build a per-topic multicast tree to disseminate events published on the topic.

2.3.3 Chord

Chord [7] nodes are assigned random IDs with t bits using a consistent hash function like SHA-1 on their IP addresses, with t being large enough so that the probability of two nodes having the same ID is negligible. When querying, a key is used for the query to find some content. Keys also have IDs with the same size t by using a consistent hash function on the key itself. IDs are ordered in a identifier circle modulo 2^t known as the *Chord ring* just like in Figure 2.5 where $t = 6$. The successor of key k (also denoted as *succ(k)*) is the node with equal ID k or the node with ID n that follows k in the identifier circle. In Chord, nodes are responsible for a key if they are the successor of that key. As we can see from Figure 2.5, key 10 is at the node 14, key 24 and 30 at node 32, key 38 at node 39 and key 54 at node 56. This way, any node in the ring knows what keys should a node be responsible for depending on their ID.

Node state and Maintenance

Each node has a table called the *finger table* with t entries. Entry i in node n has the node $s = \text{succ}((n + 2^{i-1}) \bmod 2^t)$, $1 \leq i \leq t$. In our example, the finger table of the node 8 should look similar to Figure 2.6, with the right column representing which node ID and contact is stored in the entry and the left column representing which ID is the entry's node successor to. The *finger table* is mostly used for acceleration of lookup since as long as each node knows just its own successor, it is guaranteed that a query will reach its destination.

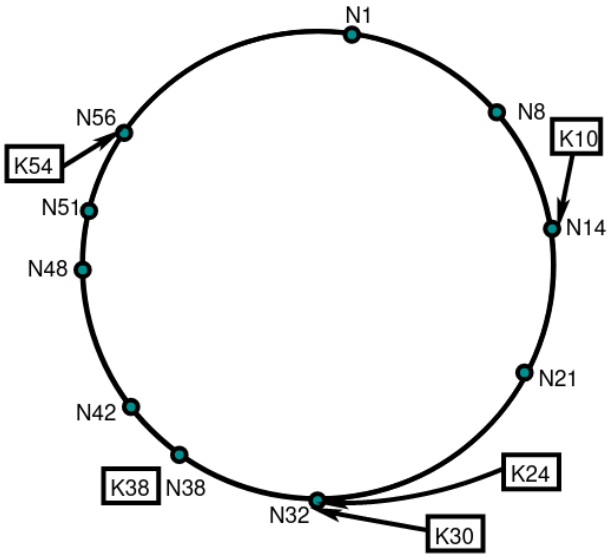

Figure 2.5: Illustration from [7]. Chord ring with nodes labeled as N(ID) and keys labeled as K(key). Arrows show what node stores the respective key.

N8 +1	N14
N8 +2	N14
N8 +4	N14
N8 +8	N32
N8 +16	N32
N8 +32	N42

Figure 2.6: Adapted from [7]. Finger table for node 8.

As for maintenance and stabilization, Chord needs to ensure each node knows its own correct successor for lookups to execute correctly . It achieves this by each node running a periodic procedure that checks and updates the finger table and successor. These procedures are *stabilize()*, *fix_fingers()* and *check_predecessor()*.

When a node *n* starts, it either starts a new *Chord ring* or joins an already existing *Chord ring*. When a new Chord ring is to be created and initialized, a node *n* starts with a non defined predecessor and sees itself as its own successor. When a node wants to join a *Chord ring*, a node *n* must know another node *n'* that already participates in the *Chord ring* and queries that node to find its own successor. To update or correct the finger table, a node *n* executes periodically *n.fix_fingers()*, iterating each finger, querying itself (and eventually other nodes) and updating each finger. To update and correct the successor, a node *n* periodically executes *n.stabilize()*, asking its successor for the successor's predecessor *p*, and deciding whether *p* should be *n*'s successor instead. If a

new node has joined the system, the successor would be updated. And finally, n notifies its successor of n's existence, potentially changing the successor's predecessor.

Lookup

In Chord, since nodes do not have enough information to directly determine the successor of any arbitrary key, two lookup procedure are used for message routing that each node can execute: *find_successor(id)* and *closest_preceding node(id)*. Since to find the node that stores specific content with key *k* is to find the successor of *k*, *find_successor(id=k)* is used for queries. As we can see in Figure 2.7, if *id* is between the node n (the node that has

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    n' = closest_preceding_node(id);
    return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
  for i = m downto 1
    if (finger[i] ∈ (n, id))
      return finger[i];
  return n;
```

Figure 2.7: pseudo code of lookup procedures from [7].

received the query) and its successor, the procedure is finished and the successor of n is returned. Otherwise, the node n starts the procedure *closest_preceding node(id)* and searches its *finger table* for the node with closest ID to *id* that precedes *id*. This search helps to find a node that is closer to the actual successor of *k* and is more likely to have the contact of that successor. As an example, consider the Figure 2.5 and let us suppose that node 8 wants to find the successor for key 54. First, node searches its finger table to find that the highest node preceding 54 is node 42 and passes on the query to that node. Node 42 then searches in its own finger table for the highest node preceding 54, finds node 51 and passes the query on. Finally, node 51 notices that its own successor is the successor of 54 and will return as answer its successor.

Strengths and Drawbacks

Chord's main weakness is tied to the structure itself, if the circle is broken for some reason, i.e. *Chord ring* is divided into two disjointed *Chord rings*, then queries may not be delivered to their destination. This is due to the fact that Chord always operates on the assumption that the identifier circle is either completely up to date or in the process of updating itself correctly, which may lead to queries with nothing to return even if the content exists in the network.

Chord's main strength is its lack of complexity overall. Chord provides a lookup function (*lookup(k)*) that yields the IP address of the node responsible for the key *k* and notifies on each node the upper layer of changes in the set of keys the node is responsible for. Besides the simple interface, the implementation of its lookup and maintenance procedures are reasonably easy to understand.es

Variants

T-Chord

T-Chord is a Chord variant that uses the T-Man algorithm to jump-start Chord more efficiently from scratch [20]. By using the T-Man algorithm, T-Chord can jump-start with many already online nodes quickly while also trying to create an optimal chord ring structure. T-Chord also uses *"leaves"* to store the *l* nearest successors to improve message delivery in case of failures and to protect the chord ring from partitioning into disjoint rings[20]. Unfortunately, T-Chord does not guarantee when the T-Man algorithm must stop since it cannot guarantee convergence has been reached.

Koorde

Koorde uses Bruijin graphs to forward lookup requests from Chord with O(log N/ log log N) hops per lookup request with O(log N) neighbors per node in N node network [15]. It achieves this due to each node m storing its own successor and list of predecessors of 2m - O(log n / n), with a trade off of degree to hop-count.

2.3.4 Kelips

Kelips [21] consists of *k* virtual *affinity groups*, numbered from 0 to *k*-1, with *k* being a parameter. Nodes are distributed among all *affinity groups* with the use of cryptographic hash functions like SHA-1. Kelips itself has not been use in practice or as base of another system or variant, which is why there will be no variants explained in this section.

Node state and maintenance

Kelips has 3 main states: Affinity group view, Contacts, Filetuples. Affinity group view is a partial set of nodes of the node's Affinity group, Contacts are the sets of constant size of nodes in other Affinity groups besides the current node's, and Filetuples is a partial set of tuples, each with a filename and a node's IP address of the same Affinity group that stores the file.

In a network of *N* nodes with *k affinity groups*, all three states are refreshed periodically within and across all groups. Each entry for all states stored at a node has an integer heartbeat count associated with it. If a heartbeat count has not been update over a pre-specified time-out period, the entry is removed. These updates originate from the responsible node

of the entry (the node the entry points to) and are disseminated through a P2P Epidemic or Gossip based Protocol.

Once a node receives a piece of information to be multicast, either from another node or application, the node gossips for a number of *rounds*, where a round is a fixed local time period at the node. Each round the node chooses a small constant-sized set of nodes from its own *affinity group* and sends them a copy of the information. This way, the protocol transmits the multicast to all nodes with high probability. Target nodes are chosen based on round trip time estimates, preferring nodes that are closer with less latency. A few of these target nodes need to be outside the node's *affinity group* to keep entries of the node's *affinity group* from expiring in other groups. Gossip messages carry several filetuple and membership entries, including new ones, recently deleted ones and ones with updated heartbeat count. Participating nodes also ping a small set of nodes they know periodically to obtain and update response times that are included in round trip time estimates. When hearing of new contacts when the contact entry set is full, the farthest node is chosen as the victim for eviction, according to the round trip estimates.

Since Kelips limits bandwidth at each node, not all entries from the states are packed into a single gossip message. Maximum quotas are applied to all types of entries that can be put into a gossip message. For each type, the quota is subdivided equally for fresh and older entries. Entries are chosen uniformly at random and unused quotas are filled with older entries.

When a node wants to join the system it must have a contact of a node that is already participating, just like other protocols. That contact returns its view to be used by the new node so it can start disseminating messages to let other nodes know about the new node's existence.

Lookup

Let us consider that node P wants to fetch a given file. Node P maps the filename to the appropriate *affinity group* by using the same hashing function used to assign *affinity groups* to nodes. P then sends a lookup request to the topologically closest contact it knows from that affinity group. When receiving the lookup request, the node searches among its filetuples for the node responsible for the file and replies with the address if that node, making Kelips lookup time O(1) with message complexity of O(1). Storing a new file works in a similar fashion. Instead of a lookup request, a insertion request is sent to the topologically closest node in the appropriate *affinity group*. When receiving a insertion request, the node chooses a node randomly from its affinity group and forwards the insertion request to that node, making that node responsible for storing the file.

Strengths and drawbacks

Kelips main weakness is maintenance overhead. Nodes will always disseminate messages despite node and delivery failures, sending messages that essentially do not contribute directly to the protocol until the next node failure. However, Kelips provides O(1) lookups with a O(1) message complexity and single hop. This is a huge advantage compared to other protocols we have seen that usually provide a lookup at a logarithmic time.

2.3.5 Kademlia

Kademlia [3] node IDs and keys are opaque and belong in a 160-bit key space. In Kademlia, there is a notion of distance between node IDs and Keys using the XOR metric. This metric is simply the integer value of the resulting binary from a XOR between IDs and keys. For example, the distance between node 1111 and node 1001 is 6 because 1111 $\oplus$ 1001 = 0110 = 6 (can also be represented as d(1111,1001) = 0110 = 6). This XOR metric offers some obvious properties such $d(x,x) = 0$, $d(x,y) > 0$ *if* $x \neq y$ and $\forall x,y : d(x,y) = d(y,x)$. XOR also has the triangle property: $d(x,y) + d(y,z) \geq d(x,z)$. XOR is also unidirectional, this means for any given x and distance $z > 0$, there is only one y such that $d(x,y) = z$. This property ensures all lookups for the same key k will always converge on the same path, regardless of the originating node. Thus, caching (key, value) pairs along the lookup path can alleviate hot spots.

Kademlia's nodes are treated as leaves in a binary tree, with each node's position determined by the shortest unique prefix of its ID. As we can observe in the Figure 2.8, the node with 0011 position is determined by the path we take to find node starting from the root. Going left in the tree means adding 1 to the prefix and going right means adding 0.

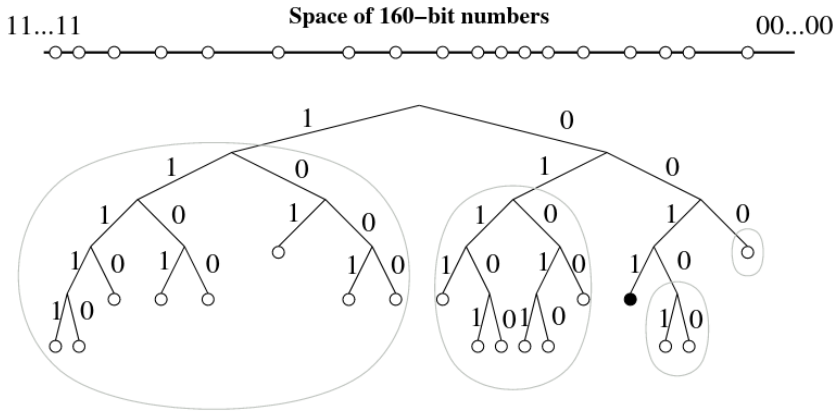

Figure 2.8: Illustration from [3]. A Kademlia binary tree. The black dot shows the position of node 0011... in the tree. Gray ovals show the subtrees the node 0011... must have a contact.

The black dot shows the position of the node with prefix 0011 and the gray ovals indicate which subtrees the node with prefix 0011 must have a contact with. The binary tree grows

by dividing into a series of lower subtrees that don't contain the node itself. Kademlia ensures that every node must know the existence of at least one node in each subtree(that has at least one node) of its binary tree. This way, it is guaranteed that any node can find any other node by its ID.

Node state and Maintenance

Kademlia nodes store each others contact information in list of (IP Address, Port, Node ID) triples for nodes of distance between 2^i and 2^{i+1} from their own ID, for each i in $0 \leq i < 160$. These lists are called k-buckets. Each k-bucket is ordered by having the least recently seen nodes at the head and most recently seen nodes at the tail. For smaller values of i, k-buckets will most likely be empty since not many nodes are appropriate if at any. But for larger values, lists can grow up to size k, where k is system-wide replication parameter for Kademlia. Choosing k should ensure that any given k nodes are very unlikely to fail within a hour from each other(for example $k = 20$).

Each node u starts out with a binary tree with one k-bucket that covers all of 160-bit space, when it reaches size k the bucket is split into two and two binary tree nodes are created, each with a new k-bucket with a new prefix. The k-buckets position in the tree represents the prefix that the nodes inside the k-bucket have. When a k-bucket is full and node u learns a new contact, if the bucket contains node u's ID then the bucket is split.

In case of an unbalanced binary tree, Kademlia guarantees that the smallest subtree around the node u's k-bucket has at least k contacts, even if this means splitting a k-bucket where node u's ID does not reside. Let us consider a new node u with prefix 000 is joining a network filled with more than k nodes with prefix 001. Every node with prefix 001 would have an empty bucket which node u will be inserted but node u would have only have k of those nodes. So a split happens in node u's k-bucket with prefix 001.

As for maintenance, Kademlia nodes use any received message as an opportunity to update and maintain contact information from other nodes. When receiving a message from another node, the appropriate k-bucket is updated. If the sending node is already in the appropriate k-bucket, then the recipient node moves the sender to the tail of the list. If the k-bucket is not full and the sending node is not already in the bucket, then the recipient node just inserts the new sending node at the tail. However, if the bucket is full, the recipient node places the sending node in a *replacement cache* of nodes that will replace stale nodes. The next time the recipient node sends a message, any unresponsive nodes get replaced by nodes in the *replacement cache*, ordered by most recently seen first. When the *replacement cache* is empty and a k-bucket is not full, if nodes fail to reply 5 times they are flagged as stale. This helps to ensure that if a network connection goes down temporarily, nodes will not isolate themselves and empty out all of its contacts.

K-buckets essentially implement a least recently seen policy except that live nodes are never removed from a list. This is due to older nodes are more likely to remain alive than

recent nodes. By preferring to keep old nodes instead of discarding them, maximizes the probability that nodes in k-buckets will remain online. This also helps to mitigate certain DoS. Flooding the system with new nodes will not remove nodes from the routing table, since Kademlia nodes will only insert new nodes in their k-buckets when old nodes leave the system.

Since Kademlia's lookup procedure is used for many maintenance purposes, the lookup section will elaborate more on how Kademlia keeps node state and (key,value) updated and persistent.

Lookup

There are four RPCs in Kademlia: *PING*, *STORE*, *FIND_NODE* and *FIND_VALUE*. The *PING* RPC is used to check if nodes are online, although it can just be piggy-backed in RPC replies. *STORE* is used to instruct the recipient node to store a certain (key,value) pair. *FIND_NODE* RPC takes a 160-bit ID as an argument. When a node receives this RPC, it returns k (IP address, Port, Node ID) triples for the k closest nodes to the target ID that the recipient knows about. These triples may come from a single k-bucket or multiple if the closest k-bucket is not full. If the recipient does not have k nodes in total of its k-buckets, it just returns all of the nodes it knows about. For the *FIND_VALUE* RPC, it behaves like a *FIND_NODE* with one exception. When the recipient has received a *STORE* RPC for the given key, it just returns the value instead of returning the k triples of the closest nodes to the given key.

Kademlia's most important procedure is *node_lookup*, a recursive algorithm to locate the k closest nodes to a given ID. The lookup initiator starts by picking α nodes from its closest non-empty k-bucket or α closest nodes if the bucket has less than α nodes. Asynchronous and parallel *FIND_NODE* RPCs are sent to the α chosen nodes with α being a system-wide concurrency parameter, for example 3. In the recursive step, the initiator resends *FIND_NODE* RPCs to the nodes that it has received from the previous RPCs replies. The initiator does not need to way for all α nodes to reply to start resending RPCs to other nodes. Of the k nodes the initiator has learned that are closest to the target, it picks α nodes and resends *FIND_NODE* RPCs to them. If nodes fail to reply quickly, they are removed from consideration until and unless they reply. If all of the k nodes fail to reply, the initiator picks another k nodes it has node yet queried and sends the RPCs. The lookup terminates when the initiator has received all responses from the k nodes it has seen. Most operations use this procedure to function. In order to send *STORE* RPCs, the node uses this procedure to find the k closest nodes to the target. To find a (key,value) pair, the lookup is performed to find k closest nodes to the key using *FIND_VALUE* RPCs instead of *FIND_NODE*.

To ensure persistence for content stored in the system when nodes leave and new ones join, Kademlia nodes republish (key,value) pairs once an hour. When a node receives a

STORE RPC for a given (key,value) pair, it assumes that the RPC was also issued to the k-1 nodes and will not republish the (key,value) pair in the next hour. If republication intervals are not exactly synchronized, only one node will republish the a given (key,value) pair every hour. The expiration time for a (key,value) pair is exponentially inversely proportional to the number of nodes between the current node and the node whose ID is closest to the key ID. This helps avoid *"over-caching"* due to the XOR's unidirectionality leading to future searches for the same key are likely to hit cached entries before reaching the closest node. Since RPCs help keep refresh buckets, traffic of RPCs traveling through nodes is generally enough to keep buckets updated. To join the network, a node u must have a contact c that is already participating. u inserts the new contact into its bucket, performs a node lookup for its own ID and finally *"refreshes"* all k buckets by picking a random ID from each one and performing a node lookup for chosen ID. This procedure helps populate u's buckets and make other nodes know about u's existence.

Strengths and Drawbacks

Kademlia's main strength is how its maintenance is performed by normal queries. On a very busy environment, Kademlia's maintenance naturally occurs as RPCs flow from one node to others, keeping node state updated even when nodes leave and join. Kademlia's weakness also has to do with how large node state can reach. Nodes in Kademlia protocol do not discard contacts of nodes that are still active and will split k-buckets in order to store even more contacts. In a network where most nodes do not tend to fail often, each node will know about a big chunk of the network that most likely will not be used actively for queries. Of course this also depends on how high is the parameter k, a bigger k leads to bigger buckets, storing more and more contacts where a fewer would suffice.

Variants

S-Kademlia

S-Kademlia [22] main purpose is to provide security to Kademlia by preventing different types of attacks. With the use of asymmetric cryptography and crypto puzzles, it helps prevent attacks like Eclipse and Sybil attacks and verify message integrity. This variant focuses more on security of the protocol than the performance of the protocol itself.

2.3.6 Discussion

From the examples we detailed above, we can observe that some of the existing designs have a higher degree of structure than others. Degree of structure means the predictability of the nodes that should appear in neighborhood lists (e.g.:routing tables) of each node, where higher the degree of structure implies a higher amount of predictability (where in the limit it is deterministic).

On the more structured side we have Chord and Pastry. Chord has a finger table that shows what each node should know about other nodes. What content each entry should have in the finger table is already decided the moment the node has an assigned ID and joined the *Chord ring*. Routing in chord also follows a strict procedure with little flexibility if at all.

In Pastry, there are multiple states, each with their own constraints and maintenance. In the routing table, not only the IDs of nodes matter to know where they will be stored in the table but also the result give by the proximity metric. The leaf set only allows nodes whose ID is numerically closest to the current node. The neighborhood set is built with nodes that are closest to the the current node via proximity metric.

On the other side of the spectrum, we find Kelips. Kelips sense of structure is much more relaxed compared to Chord's and Pastry's. Each node has three states, each one having simple constraints. Kelips's affinity group view is a partial view of all nodes in the current node's affinity group. Contacts are the constant size sets of of nodes from each affinity group different from the current node's. And finally, filetuples is a set of pairs of keys and nodes' contact information, associating keys with nodes for all nodes in the current node's affinity group.

We can also deduce that Kademlia's position in the spectrum lies near the middle, leaning towards the structured side. Kademlia's binary tree can hold nodes with many varied node IDs depending on the ID distribution, network size and the parameter k. Maintenance is mainly performed passively, as messages with queries are routed to nodes. This resembles the low maintenance property that unstructured overlays possess.

In this thesis, we aim our solution to be in the middle of the spectrum. Having as much properties from structured overlays as with unstructured ones, providing more benefits and less drawbacks than a design near the extremes of the spectrum. This design will be introduced and discussed later in Chapter 3.

2.4 Load balancing in peer-to-peer systems

Considering the discussion in previous sections, we can see that many examples of DHTs do not consider a node's capacity as part of any load balancing strategies. All of them assume node capacity to be the same for every node that participates or joins. This becomes a real problem when node capacity is not as homogeneous among nodes as these examples assume to be. Nodes with lower capacity may end up overloaded, potentially failing at worst, and at best, queries that pass through these nodes will be held up longer, increasing latency. In contrast, nodes with higher capacity, end up not being used to their full potential, wasting resources and potentially missing opportunities to improve overall system performance.

In the work [23], a similar problem was discussed and the target for their solutions was

unstructured overlays. This work acknowledges the heterogeneity of nodes in unstructured overlays and provides solutions at the service and membership layer to generate a load distribution scheme where nodes that are more powerful contribute with more resources, by having additional overlay neighbors.

Although there are more solutions like for example Chunky [24], these solutions are for unstructured overlays. Translating solutions meant for unstructured overlays for structured overlays is not a trivial step, particularly because of the reuse of neighbors affect the connective properties of DHTs. This means that these techniques may break properties of DHTs, losing guarantees and potentially other provided benefits, losing performance compared to not using a load balancing solution at all.

Discussion

The heterogeneity of capacity among nodes in P2P systems can be an important topic to discuss. If capacity is disregarded, nodes may fail and/or overall performance may decrease because of this. Solutions for similar problem we present in this work exist, although they are not appropriate for structured overlays and are not easy to translate into this class of solutions.

2.5 Summary

In this chapter, we explained the essential concepts of P2P systems, their applications and services. We also introduced the concept of overlays and their types, focusing more on structured overlays.

We started by explaining unstructured overlays, their high fault tolerance and dissemination of messages due to their lack of strict topology. We explored some examples of these overlays and briefly introduced them and their most interesting and relevant traits.

Next we introduced structured overlays and DHTs and how they work with *identifiers* and keys to locate resources among participants in the system. We also talked about how overlays are evaluated by explain some of the popular metrics used to evaluate performance of unstructured or structured (and sometimes both) types of overlays.

We studied what are DHTs and deeply explain some of the most popular DHTs that are still based upon or used today in modern applications and services. All of the discussed DHTs do not implement load balancing that is aware of a nodes capacity. Instead, they assume every node is equal when it comes to capacity.

And finally, we discussed load balancing in peer-to-peer systems, more specifically in DHTs, and how non-trivial it is have load balancing solutions from unstructured overlays translate int to structured overlays.

We aim to provide a solution that is in the middle of the design space of DHTs, while

also providing fair load balancing aware of each node's individual capacity. Further details will be discussed in Chapter 3.

3

Proposed Solution

We have seen various advantages and disadvantages of structured and unstructured overlays. Although mixes of both technically exist, those usually only have subtle changes that bring a small contribution to the overall protocol. However, all of them seem to lack awareness of individual node capacity. In this paper we propose a true hybrid overlay, in the middle of the design spectrum, providing load balancing aware of each node's own capacity.

In Section 3.1, we will describe our solution and its approaches; in Section 3.2, we will explain what metrics we use to evaluate the performance of our solution and what it will be compared to; and finally, in Section 3.3, we will present the division of work and its schedule.

3.1 Our solution

Our solution will be based on the IPFS' [6] implementation of Kademlia, more specifically libp2p's [4] peer routing implementation. In our implementation of Kademlia, the parameter k will be replace by two fields that each node will have: *k-in* and *k-out*.

k-in will represent a node's *in-degree*, limiting the number of nodes that can have the current node as a contact. The field *k-out* will represent a node's out-degree, limiting the amount of nodes' contact information the current node can know. These fields will be useful to influence how nodes know each other based on their capacity. A higher capacity node should have a higher *k-in* and *k-out*, since the more nodes know of its existence, the more queries will be routed for this node, increasing query efficiency. Lower capacity nodes should have a lower *k-in*, since it decreases the likelihood of many queries passing through the node that could potentially overload it.

What we expect to gain from our solution is better use of available capacity of nodes, thus increasing overall performance specially in latency of requests.

Node capacity depends on the resources available when the node starts. Although, capacity may change even when a node is participating in systems. Because of this, our

solution is divided into two approaches: static approach and the dynamic approach.

3.1.1 Static approach

In this approach, we will focus more on the properties of the node capacity that do not change over time (e.g.: processing power and total bandwidth). Since capacity will not change for nodes over time, initial values of *k-in* and *k-out* will be pre-define by the user according to node capacity.

3.1.2 Dynamic approach

In this approach, we will focus more on the properties of the node that do change over time (e.g.: latency and available bandwidth). Since capacity can change over time, the focus of this approach will be the dynamic estimation of capacity and how to dynamically influence *k-in* and *k-out* based on that estimation. This estimation of node capacity will be performed using an implementation of ResEst [25].

3.2 Evaluation

To evaluate our solution fairly, we will use some metrics described in Section 2.2.3 to evaluate performance of both of our approaches, compare them to the original implementation of IPFS' [6] Kademlia and our own implementation of Koorde [15], and finally perform experiments in a emulated environment, simulating resources and network traffic, and therefore capacity for nodes. All implementations will be developed to work in the environment of libp2p [4], using containers to run implemented code. This way, every implementation of we develop and libp2p's implementation will be on equal playing field and our metrics will be fair.

The metrics that we will use are the following: maintenance overhead, latency and fault tolerance.

3.3 Work Schedule

In this section we present the work schedule in the form of a Gantt chart in Figure 3.1. The work will be divided into the following tasks:

Design of static approach and its validation in this task we will design, implement and validate our static approach.

Design of Koorde and its validation in this task we will implement and validate our Koorde [15] implementation.

Design of dynamic and its validation in this task we will design, implement and validate our dynamic approach.

Complete evaluation of approaches we will evaluate both approaches to our Koorde implementation and the original libp2p's Kademlia implementation using the metrics mentioned in the section above.

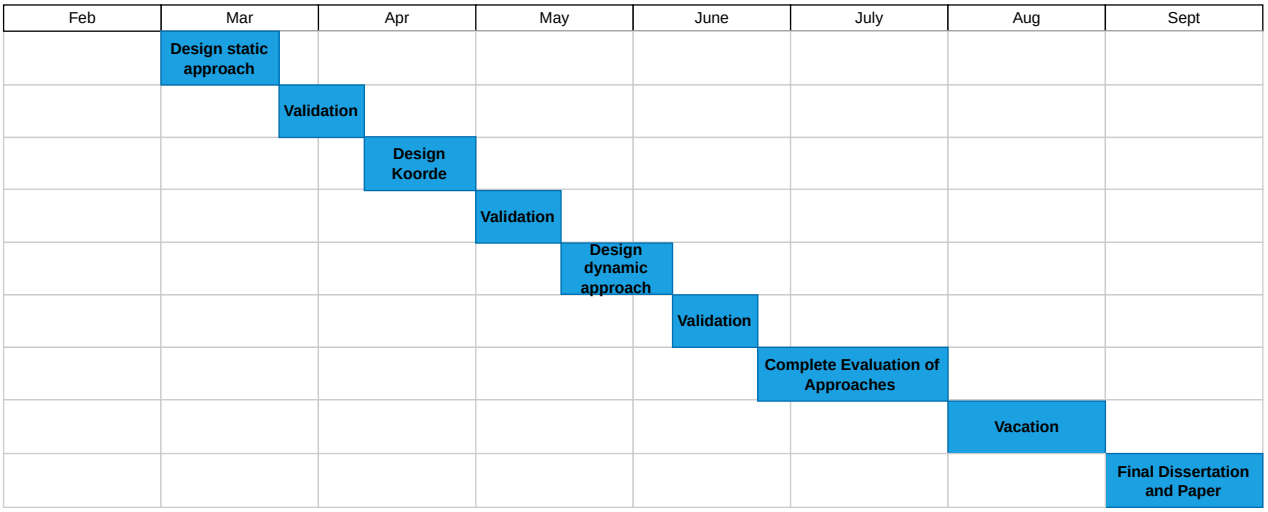

Figure 3.1: Work schedule in the form of a Gantt chart.

Bibliography

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

[2] J. Leitão. *Topology Management for Unstructured Overlay Networks* (cit. on pp. 1, 5).

[3] P. Maymounkov and D. Mazières. *Kademlia: A Peer-to-peer Information System Based on the XOR metric* (cit. on pp. 2, 21).

[4] *libp2p: A modular network stack*. URL: https://libp2p.io/ (cit. on pp. 3, 28, 29).

[5] P. Labs. *Filecoin: A Decentralized Storage Network* (cit. on p. 3).

[6] J. Benet. *IPFS - Content Addressed, Versioned, P2P File System* (cit. on pp. 3, 28, 29).

[7] I. Stoica et al. *Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications* (cit. on pp. 4, 10, 16–18).

[8] J. Roskind. *QUIC:MULTIPLEXED STREAM TRANSPORT OVER UDP* (cit. on p. 5).

[9] *Napster*. URL: https://napster.com/ (cit. on p. 5).

[10] *Gnutella*. URL: https://web.archive.org/web/20080525005017/http://www.gnutella.com/ (cit. on p. 5).

[11] J. Leitão, J. Pereira, and L. Rodrigues. *HyParView: a membership protocol for reliable gossip-based broadcast* (cit. on p. 6).

[12] D. G. Spyros Voulgaris and M. van Steen. *CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays* (cit. on pp. 7, 8).

[13] A.-M. K. Ayalvadi J. Ganesh and L. Massoulié. *SCAMP: Peer-to-peer lightweight membership service for large-scale group communication* (cit. on p. 7).

[14] S. R. Yatin Chawathe et al. *Making Gnutella-like P2P Systems Scalable* (cit. on p. 7).

[15] M. F. Kaashoek and D. R. Karger. *Koorde: A simple degree-optimal distributed hash table* (cit. on pp. 8–10, 19, 29).

[16] P. F. Sylvia Ratnasamy et al. *A Scalable Content-Addressable Network* (cit. on pp. 10, 11, 13).

[17] P. Francis. *Yoid: Extending the Internet Multicast Architecture*. URL: http://www.aciri.org/yoid/docs/index.html (cit. on p. 11).

[18] A. Rowstron and P. Druschel. *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems* (cit. on pp. 13, 14).

[19] P. D. Miguel Castro, A.-M. Kermarrec, and A. Rowstron. *SCRIBE: The design of a large-scale event notification infrastructure* (cit. on p. 16).

[20] M. J. Alberto Montresor and O. Babaoglu. *Chord on Demand* (cit. on p. 19).

[21] K. B. Indranil Gupta et al. *Kelips : Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead* (cit. on p. 19).

[22] I. Baumgart and S. Mies. *S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing* (cit. on p. 24).

[23] V. H. Menino. *A NOVEL APPROACH TO LOAD BALANCING IN P2P OVERLAY NETWORKS FOR EDGE SYSTEMS* (cit. on p. 25).

[24] K. Y. Vidhyashankar Venkataraman and P. Francis. *Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast* (cit. on p. 26).

[25] P. A. C. Vítor Hugo Menino and J. Leitão. *ResEst – Algoritmo Distribuído para a Inferência de Recursos da Rede* (cit. on p. 29).