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

June, 2023

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: Doutor João Carlos Antunes Leitão
Associate Professor

Examination Committee

Chair: Doutor Hervé Miguel Cordeiro Paulino
Associate Professor, Faculdade de Ciências e Tecnologia da UNL - Dep. De Informática.

Rapporteur: Doutor João Nuno de Oliveira e Silva
Assistant Professor, Instituto Superior Técnico, Dep. Eng. Electrotécnica e de Computadores.

Adviser: Doutor João Carlos Antunes Leitão
Associate Professor, Faculdade de Ciências e Tecnologia da UNL - Dep. De Informática

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
June, 2023

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

Nos sistemas ponto-a-ponto onde nós não têm vista total do sistema, não é trivial distribuir carga de acordo com as capacidades de cada nó individual. Estratégias de balanço de carga de acordo com a capacidade de cada nó individual podem não só melhorar a eficiência para nós com maior capacidade, como também evitar sobrecarregar nós, potencialmente evitando algumas falhas. Para sobreposições não estruturadas existem este tipo de estratégias, mas a sua tradução para sobreposições estruturadas não é trivial. Estas estratégias podem quebrar propriedades, perdendo garantias e potencialmente outros beneficios, piorando o desempenho no sistema em de melhorar.

Nesta tese, propomos uma variante de Kademlia, que usa uma estratégia de balanceamento de carga ciente das capacidades dos nós do sistema e que situa-se no meio do espaço de desenho em termos de estrutura.

Palavras-chave: sistemas ponto-a-ponto, tabelas de dispersão distribuídas, balanceamento de carga 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 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 less explored 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 Contributions

The main contributions we generated 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 that is estimated using an implementation ResEst [4] combined with a implementation of HyParView [5].
- An experimental comparison between the aforementioned design and the original implementation of Kademlia.

1.3.1 Research Context

The work to be performed in this thesis was 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 [6], Filecoin [7], and IPFS [8].

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 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 elaborate our proposed solution, explaining formulas and reasoning behind our implementation.
- In Chapter 4 we describe our experiments, parameters, configurations, and the environments of which we analyzed our solution and a original implementation of Kademlia, comparing the results against each other.
- In Chapter 5 we present our conclusions from our analysis and discuss what could be done from them in the future to innovate DHTs.

2

Related Work

In this chapter, we first introduce and study relevant topics and concepts that are the base for the elaboration of this thesis while also presenting related works. The chapter structure is as follows: in Section 2.1 we introduce what are peer-to-peer systems; in Section 2.1.1 we 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 briefly explore unstructured overlays and some of their examples; in Section 2.2.2 we introduce what is a structured overlay and briefly describe their most common type; in Section 2.3 we explore what is a DHT and describe in detail some of their famous examples; in Section 2.4 we discuss some solutions to load balancing in p2p systems and the lack thereof for structured overlays while presenting a Unstructured Overlay solution: ResEst; 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 [9] 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.

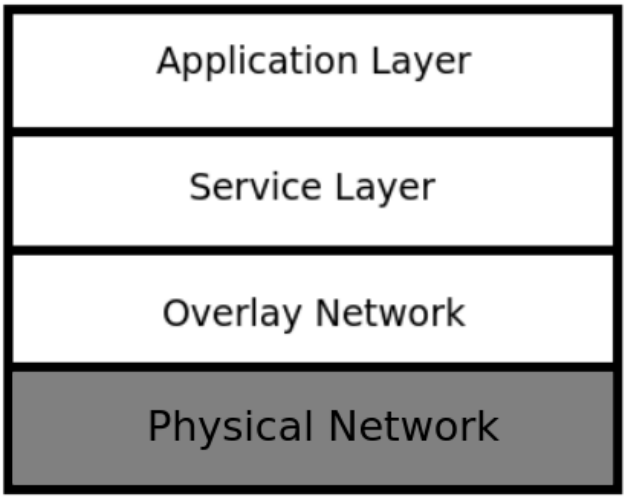

Figure 2.1: P2P Architecture adapted from [2].

Physical Network layer

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 [10].

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 [11] and Gnutella [12]. 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 [5] 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 [13], 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 [14] 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 [15] 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 [16] 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 [13]. 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 [16]. 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 [16]. 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.

Degree of structure in Structured Overlays

Usual performance metrics for structured overlays focus on hop counts, fault tolerance, load balance,degree and maintenance overhead [16] 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

Load balance

How much load does each node experience as an intermediate node and how even the keys are distributed across nodes [16]. 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.

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 [9]. 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 [17] 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

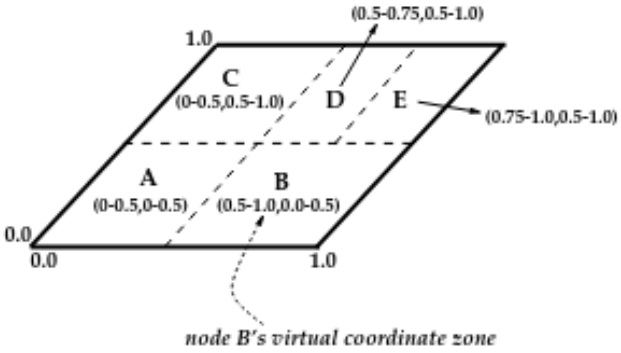

Figure 2.2: Illustration from [17]. An example 2D space with 5 nodes. Note: this is a simplified example, the reader must remember that the coordinate space wraps.

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 [18]. 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

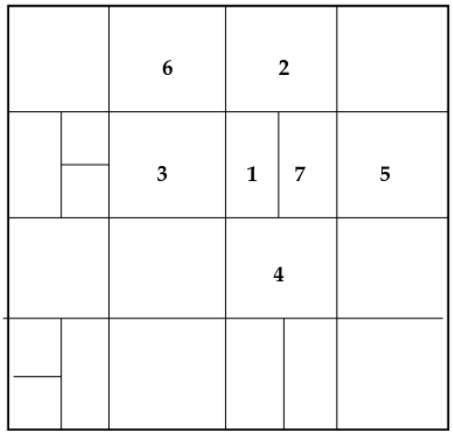

Figure 2.3: Illustration from [17]. Example 2D space with some numbered nodes and their zones.

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 [19] 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.

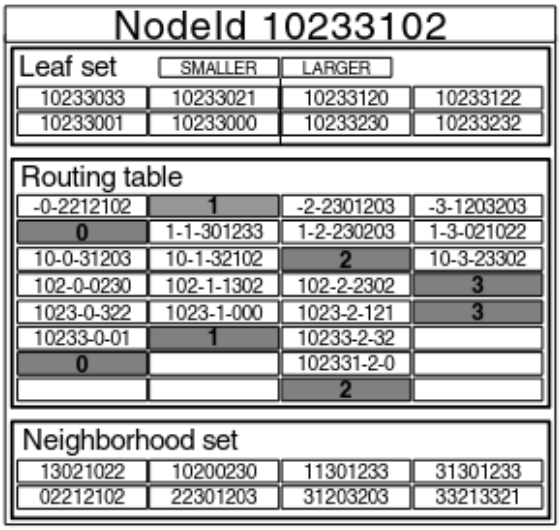

Figure 2.4: Illustration from [19]. State of a node whose ID is 10233102. $b = 2$ and $l = 8$. All numbers are in base 4. Node IDs are 16 bits long. IP addresses are not shown.

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 [20] 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 [9] 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

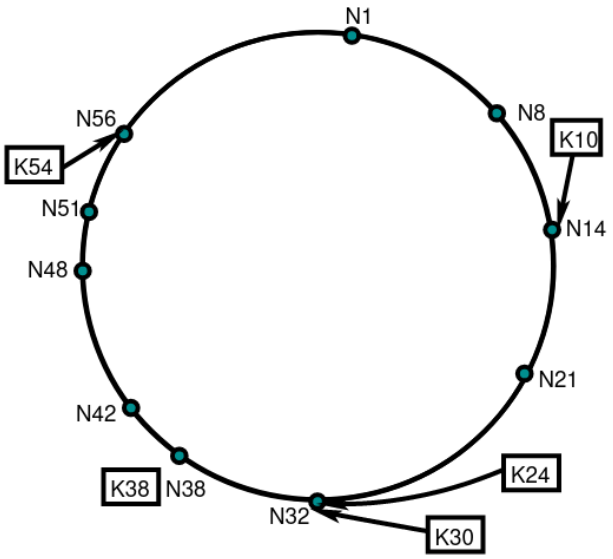

Figure 2.5: Illustration from [9]. Chord ring with nodes labeled as N(ID) and keys labeled as K(key). Arrows show what node stores the respective key.

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

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

Figure 2.6: Adapted from [9]. Finger table for node 8.

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.

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 [9].

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 [21]. 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[21]. 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 [16]. 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 [22] 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.

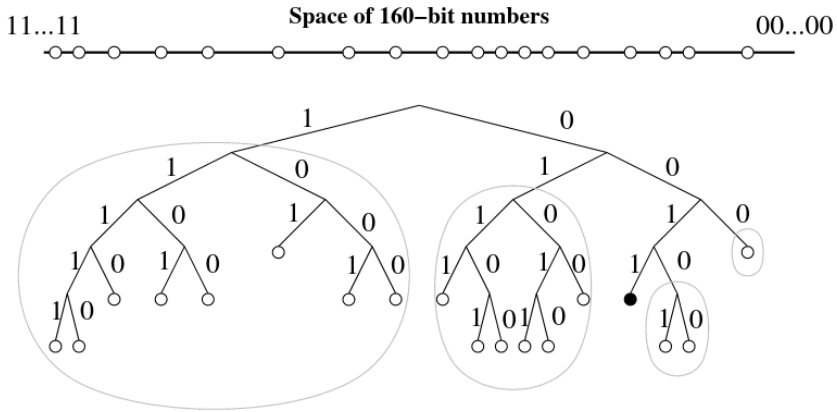

Figure 2.8: Illustration from [3]. A Kademlia binary tree. The black dot shows the position of node 0011... in the tree. Gray ovals show the subtrees the node 0011... must have a contact.

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.

Our solution will be based on the Kademlia protocol and it can be considered a variant. More will be discussed in Chapter 3.

Variants

S-Kademlia

S-Kademlia [23] 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.

Nodes are generate public and private keys to make node assignment secure and protect the integrity of the sent messages between other nodes. Signatures are split into two categories: *Weak signature* and *Strong signature*.

Weak signatures protect the IP address, port and a timestamp that specifies how long the signature is valid, preventing replay attacks. This type of signature is mainly used in *FIND_NODE* and *PING* RPCs, since the integrity of these types of messages is dispensable.

Strong signatures protect the whole message, ensuring integrity of the message and increased protection against Man-in-The-Middle attacks. Nonces inside RPC messages provide protection against replay attacks.

Besides these signatures to protect message integrity, S-Kademlia has two other ways to protect itself from Eclipse and Sybil attacks: *Supervised signature* and *crypto puzzle signature*.

When a signature's public key is signed by a trusted certificate authority, the resulting signature is a *Supervised signature*. This signature is needed during the bootstrapping phase where few nodes exist in the network, preventing Sybil attacks.

When having a trusted certificate authority is not readily available, a crypto puzzle is used to prevent Eclipse and Sybil attacks.

Since our solution barely changes anything (in-degree of others nodes change based on a probability), Kademlia and its variants can be implemented along side our solution without breaking any assumptions that Kademlia and its variants rely on.

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 aimed our solution to be in the middle of the spectrum. Our solution will only affect in-degree of nodes using a probability. This affects which nodes know other nodes without a concrete restriction. This steers Kademlia into having more properties from unstructured overlays, 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 [24], 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 [25], 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.

Even when variants of DHTs that we discussed exist to improve upon the original design focusing on one aspect, they do not focus on load balancing nodes according to their capacity:

S-Kademlia focuses on message integrity and prevention from attacks by using public/private key signatures.

T-Chord helps start Chord from scratch more quickly,improves message latency and helps prevent creating disjoint partitions using the T-Man algorithm and a successor

list.

Koorde reduces the number of hops per lookup request with a trade off of degree to hop-count.

Scribe builds a publish/subscribe service, giving Pastry a useful use case besides being a DHT.

2.4.1 ResEst

ResEst [4] is decentralized algorithm with the intention of estimating node capacity in the network and show an individual node where it inserts itself according to its own capacity. In order to create a estimation, a message is sent on a Random Walk containing a histogram and average of node capacities of the nodes in the network. Each time it reaches a node, it identifies where its class on the histogram is and increments it. The node then calculates the *Margin of Error* and if it divided by the average is below the acceptable *margin of error* (a configurable parameter) the node sends back the message to its originator.

ResEst assumes it works along side a unstructured overlay that provides memberships where every node has roughly the same amount of neighbors like HyParView [5]. Using this algorithm in structured overlays will result in bad quality estimates that cannot be used in a load balancing strategy.

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 a 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. Even when variants of the protocols we presented in Section 2.3 exist and solve some of the problems of the original design, none of them address load balancing

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 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

3.1 Motivation for node Aware Load Balancing

As we explored Load Balancing in P2P systems in Chapter 2 in Section 2.4, load balancing strategies that are aware of each node's individual capacity for Structured Overlays are scarce. Although some strategies exist for Unstructured Overlays, these do not apply to structured overlays as they tend to break assumptions and properties essential for Structured Overlays, resulting in potential loss of benefits and loss of overall performance.

Furthermore, the lack of such strategies for Structured Overlays may overload weaker nodes or waste unused resources for more powerful nodes, potentially wasting a performance increase.

The goal for this thesis is to devise a load balancing strategy aware of node's individual capacities for Structured Overlays. In order to achieve our goal, it would be useful to be able to estimate the resource distribution for the whole network so that each node can act upon that estimation, comparing it with its own resources. Instead of developing a new resource estimation algorithm from scratch, we decided to use an already existing algorithm, ResEst [4]. In combination with the membership protocol HyParView [5], every node is able to make good quality estimations of the distribution of capacity in the network.

3.2 Solution

3.2.1 Overview

Our devised solution is a variant of a structured overlay with a load balancing strategy that is aware of available capacity of each node and reacts accordingly. The structured overlay we chose as base was Kademlia [3], a DHT protocol.

Our load balancing strategy works best in a decentralized network that is heterogeneous in network capacity. The strategy is based on the ability to change the in-degree of nodes according to a node's capacity compared to the rest of th network. If node n has in-degree

x, x nodes in the network know the contact information of n. The higher n's is , the more nodes know of n, and increasing the likelihood of messages flowing throughout the network eventually reaching n. This greatly influences n's contribution of its resources towards the service that the system provides. By changing n's in-degree, we change the likelihood of messages eventually reaching n and, at the same time, change the how much of n's resources will contribute towards the service. If we change the in-degree of node n according to its perceived individual capacity compared to the rest of the network, we can better utilize n's resources, potentially reducing the amount of unused resources or preventing n from overloading. This strategy only works as intended if no node behaves in a byzantine way.

For our strategy to work, we need nodes to be able to compare their own individual capacity with the rest of the network and a way to change the in-degree based on the result of that comparison. To achieve this, we devised the following 3 techniques: adding a field *forgetme* in messages that dictates whether or not the sending node's contact information should be in the receiving node's routing table; classifying node resources with *Capacity Classes*, making comparing capacity between different nodes simplified; and changing the size of the *k-buckets* based on the comparison of the current node's and the network's capacity.

In order for nodes to compare themselves with the rest of the network, we use the resource estimation algorithm ResEst [4] to obtain an estimation for the distribution of capacity in the entire network. Every node will have its own estimation and make decisions upon it independently. However, in the case of ResEst, the underlying protocol must provide a network with roughly the same number of neighbors for each node. To solve this, we chose to have the HyParView [5] membership protocol working along side Kademlia to provide such condition, in order to have good quality ResEst resource estimations.

3.2.2 Architecture and System model

Our solution has two main protocols working independently and simultaneously: HyParView and Kademlia. HyParView provides membership for the ResEst algorithm which itself provides estimations of node capacity of the network. These estimations are then used in our strategy to make decisions whenever a message is ready to be sent to another node in the network.

Furthermore, the three most important concepts that are essential for our solution to function are the following: *Capacity Classes*, the *forgetme* field and the *Capacity Class* percentile.

Capacity Classes are used by each node to compare resource capacity with the rest of the network in a independent and deterministic way. *Capacity classes* represent raw capacity of a resource like RAM, Bandwidth, CPU and others in the form of a integer. The more

powerful a resource is, the higher its *Capacity Class*. For example, node n has the resource r with a *Capacity Class* of 10 and node q has the resource r with a *Capacity Class* of 3. In this scenario, node n has a higher capacity than node q when it comes to the resource r.

The *forgetme* field is a field added to Kademlia's RPCs [1] that change the in-degree of nodes according node's relative capacity compared to rest of the network. It achieves this by informing the receiving node whether or not the contact information of the sending node should be in its routing table. For example, if node n sends a message to node q with a *forgetme* field value of true, node q will remove n's contact information from its routing table. If the *forgetme* value was false, q would add n's contact information to its routing table.

The *Capacity Class* percentile of resource r for the node n is the percentile of nodes in the network that have a lower or equal *Capacity Class* for the resource r. This is helps a node n determine where n is in a hierarchy of capacity in the network for a specific resource. For example, if n's resource r has a percentile of 0.7, it means that approximately 70% of nodes in the network have the resource r with a *Capacity Class* lower or equal to the *Capacity Class* of node n's resource r.

3.2.3 ResEst

ResEst [4] is a decentralized algorithm based on random walks (a message that flows randomly throughout the network) that is used to estimate resource distribution in a network. It achieves this by building a histogram with different resource classes as each node contributes its resource information when the random walk passes through them.

When a node wants to start an estimation, it starts a random walk by sending a message to a random neighbor with the following contents: a histogram, the mean of the contributions, the number of contributions and the contact information of the node that started the random walk. Every time any node receives this message, they check the histogram for their class, increments it, updates the mean and checks the stopping condition.

Using Equations 3.1 and 3.2, we can calculate the current error rate of the histogram and the acceptable *margin of error*. These equations depend on four parameters: z - the confidence coefficient for the confidence level of the number of samples collected so far; σ - the standard deviation of the samples obtained; n the number of samples collected; and m the current mean of the samples obtained. If the error rate is below the provided maximum acceptable *margin of error*, then the node sends the message back to the originator of the random walk.

$$marginOfError = z \times (\sigma / \sqrt{n}) \tag{3.1}$$

[1] STORE RPCs and FIND_NODE RPCs that are used for a node join operation or to determine which nodes are best to store a certain value do not have this. Nodes will ignore this field to avoid bad key/node distribution in the network

$$errorRate = marginOfError / m \tag{3.2}$$

Intuitively, a high standard deviation will be amortized the more samples are collected. In uniform distributions of capacity, fewer samples are needed in order for the random walk to end, since samples will tend to have a lower standard deviation even with fewer samples. However, in more heterogeneous networks, more samples are needed to compensate the high standard of deviation. This means the stopping criteria depends on the quality of the sample rather than the size of the network.

In our implementation of ResEst, each random walk has multiple histograms, each for a resource r we want to use for our estimation of capacity in the network. Each histogram class is a *Capacity Class* and a random walk ends when all histogram errors are below the *margin of error*.

ResEst [4] assumes that the values being estimated are static values, never changing over time. However, in order for ResEst to be useful for our solution, we need it to estimate values that change over time, depending on how resources are being used. To take into account the dynamic nature of the values we are trying to estimate, the algorithm is performed periodically over time (for example: every 60 seconds), sending a message on a random walk even if the last random walk has not returned to its originator.

Furthermore, we have a sliding window of estimation samples to calculate a weighted average of samples, prioritizing the latest estimation, in order to account for spikes of high or low resource usage. This window contains *Capacity Class* percentiles of the originator node for each time a random walk has returned.

Even with these modifications to the implementation and the addition of the sliding window, ResEst assumes that the underlying protocol is a membership protocol of a network in which every node has roughly the same amount of neighbors. Which our base protocol (Kademlia) does not provide. The workaround we used was to have HyParView and Kademlia working simultaneously and independently of each other. The HyParView membership will be used for ResEst to make good quality estimations exclusively and will not affect any aspect of the Kademlia protocol directly and vice-versa.

3.2.4 HyParView

HyParView [5] is a membership protocol capable of maintaining an Unstructured Overlay where every node has roughly the same number of neighbors. It has 2 partial views: *passive* and *active*.

Active view contains the current neighborhood of the node. Each node has a two way link with every node in its active view. If node n has node p in its active view, then p has n in its active view as well. Furthermore, nodes have established TCP

connections with every node in its active view. This view has a small maximum size.

Passive view contains the contact information of nodes that will hopefully replace the nodes in the active view in case of node failure. This view's maximum size is much larger than the active view.

We decided to integrate HyParView in our solution in order for ResEst to provide us with good quality estimations. ResEst assumes that it runs in a network where every node has roughly the same number of neighbors. HyParView can provide such property due to the two important strategies for maintaining its partial views: a cyclic strategy and a reactive strategy.

For the reactive strategy, if node n suspects that node p from its active view has "failed"(by actually failing or blocking), n removes p from its active view and selects a random node q from its passive view to establish a TCP connection. If the connection is not established, n considers the node as failed, removes the node from the passive view and tries another random node. If the TCP connection is established with the node q, n sends a *Neighbor* request to q with its own contact information and a priority level. If n has 0 nodes in its active view, the priority level is high, otherwise the priority is low.

When q receives the request, if the priority level is high, q adds n to its active view, removing a random node from its active view if the maximum size has been achieved. If the priority level is low, only if q has space in its active view will it add n to its active view. If the request is accepted, n and q remove each other's contact information from the passive view and put them in the active view. If the request is rejected, n repeats the whole procedure with another random node from its passive view (without removing q from the passive view).

For the cyclic strategy, every node periodically performs a shuffle operation with a peer in the network. A node n starts by creating an exchange list with the following contents:K_a nodes from its active view, K_p nodes from its passive view and n's own contact information(where K_a and K_p are configurable parameters). *Shuffle* requests are sent as a random walk with an associated *"time to live"*(which can be configured) and the created list.

When a node receives a *Shuffle* request, first it decreases its time to live. If the receiving node q has more than 1 node in its active view and the time to live is greater than 0, q selects a random node from its active view, different from the node that sent it to q and forwards the *Shuffle* request. Otherwise, q accepts the request and sends back, using a temporary TCP connection, to the originator of the *Shuffle* request, a list filled with random nodes from its passive view with the same size as the list inside the *Shuffle* request. Furthermore, both nodes integrate the nodes from each other lists in their passive views. When a passive view gets full, each node will attempt to remove the nodes that it

has sent in the list and remove randomly if all of those nodes have been removed from its passive view.

HyParView [5] serves as a membership protocol for the algorithm ResEst to be performed in a Structured overlay where nodes have roughly the same amount of neighbors [4]. Without HyParView, ResEst algorithm could not be performed correctly, resulting in poor quality estimations that would not be useful for our solution.

3.2.5 Capacity Class

The *Capacity Class* is an integer that represents a class of how much work a resource can take without compromising performance and/or becoming overloaded. A resource can be CPU, bandwidth, RAM, or any resource that can influence overall system performance. The higher the *Capacity Class* of resource r, the more work r can take without overloading. For example: a CPU with a *Capacity Class* of 3 needs less work to start overloading and slowing down its throughput than a CPU with a *Capacity Class* of 10.

This concept is useful when comparing different node's resources together by defining one simple way to represent the resource capacity with a single integer. The number of classes can be changed to fit the desired granularity for the resource capacity. For example, CPU can use 10 classes and, at the same time, bandwidth can use 20 classes. The classes for the CPU are completely independent from the classes for the bandwidth. Using low number of classes will tend to make *Capacity Classes* generalize more absolute resource capacities compared to when using a high number of classes. If we use a high number of *Capacity Classes* like 100, the range of capacity for a resource in a single class is narrower than using a low number of *Capacity Classes* like 5 or 10. In order to calculate the *Capacity Class* of a resource, we devised the Equations 3.3 and 3.4.

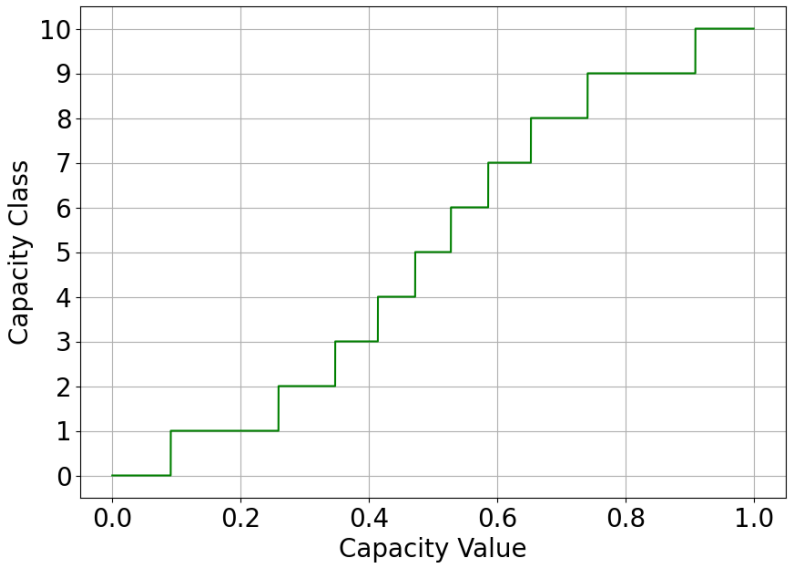

Figure 3.1: A Graph of the *Capacity Class* function. (Equation 3.4)

$$C = R \times Q \tag{3.3}$$

$$CapacityClass = \left\lfloor \frac{N}{1 + exp\left(\frac{S(R-2C)}{2R}\right)} \right\rfloor \tag{3.4}$$

***N* (Capacity Classes or Max Capacity Class)** Total number of classes or the maximum possible *Capacity Class* (includes 0).

***C* (Capacity Value)** A positive real number that represents the current capacity of the resource. The higher the *Capacity Value*, the more powerful the resource is, the higher the *Capacity Class*.

S* (Curve Steepness)** A positive real number that represents the steepness of the curve when determining the capacity class. The higher ***S is , the steeper the curve.

R* (Resource capacity)** A positive real number that should represent a resource's maximum capacity. It is also the maximum possible value ***C is allowed to be.

Q* (Resource Current Capacity)** A real number between 0 and 1 that represents the current capacity relative to ***R. if ***Q*** is 0.2, then the current capacity of the resource is 20% of ***R***.

In Figure 3.1, we can observe how the *Capacity Class* behaves when values are attributed to the the parameters and a range of input values is defined. The exact configurable parameters used in the graph are the following: $\boldsymbol{N} = 10$; $\boldsymbol{S} = 7.2$; $\boldsymbol{R} = 1$;

When choosing the appropriate value for ***S*** and ***N***, we need to consider how many *"classes"* of the same type of resource we want to have and how big the range of values of ***C*** should each *"class"* represent.

We can also observe in Figure 3.1, each class is not attributed equally when it comes to the range of values of ***C*** (Capacity Value) they represent. This is on purpose since the difference between capacity may or may not scale linearly when it comes to very powerful (or weak) node resources. When a resource gains more capacity (or or loses capacity) when it already had a lot of capacity (or very little capacity), the change is less impactful to the overall performance. When a weak resource improves its capacity (or a powerful resource decreases its capacity), the change is much more impactful to the overall performance.

3.2.6 The forgetme field

In our solution, every node gathers information on the capacity distribution of the network. Using that information, each node compares to its own capacity and makes decisions on to manipulate how many nodes in the network should know their contact information. The gathered information is stored in the form of *Capacity Class* percentiles, estimated using ResEst random walks, in a sliding window. Older random walks get discarded as new ones arrive if there is no other space left in the capacity limited window.

The absence of n's contact information in other nodes routing tables leads to less nodes knowing n exists. The less nodes know about n's existence, the lower the likelihood of queries or messages reaching node n, reducing n's workload in the network. On the other hand, the more nodes know n's contact information, the more nodes know the existence of n. The more nodes know the existence of n, the higher the likelihood of queries or messages reaching n, increasing n's workload in the network.

To control the presence or absence of contact information in other nodes' routing tables, we added a field named *forgetme* that can have two possible values: $\boldsymbol{true}$, which instructs the receiving node that the sending node's contact information should not be in its routing table, and $\boldsymbol{false}$, which instructs the receiving node that the sending node's contact information should be added to its routing table. Every message a node sends has the *forgetme* field.

By using Equation 3.5, we can calculate the probability of a message having a *forgetme* field value of $\boldsymbol{true}$ for a single resource $\boldsymbol{r}$. Every time a ResEst random walk finishes and returns to the original owner, the owner will recalculate this probability, with the new provided estimation.

$$Probability = (1 - p_r) \times P \tag{3.5}$$

p_r (Percentile of the current node's *Capacity Class* for resource r) A number between 0 and 1 that represents the *Capacity Class* percentile for resource $\boldsymbol{r}$. This can be estimated using the samples gained from the ResEst algorithm.

P (Max Probability) A real number between 0 and 1 that represents the maximum probability any node can reach for determining the value of *forgetme* field for any resource.

To calculate $\boldsymbol{p_r}$ using the estimation samples in the sliding window, we use the Equation 3.6.

$$p_r = \begin{cases} \dfrac{p_{r1} \times w_h + \sum_{i=2}^{W} p_{ri} \times w_l}{w_l \times (W-1) + w_h} & , W > 1 \\ p_{r1} & , W = 1 \\ 1 & , W = 0 \end{cases} \tag{3.6}$$

p_{ri} **(Estimation for resource r with age i)** The estimation for the percentile of nodes with less or equal *Capacity Class* for resource r with age i. The higher the i the older the estimation is.

W **(Current Sliding Window size)** The current number of available estimations in the sliding window.

w_h A number between 0 and 1 to represent the weight of relevance for the most recent sample. The higher the number, the bigger the relevance.

w_l A number between 0 and 1 to represent the weight of relevance for all samples after the first one. The higher the number, the bigger the relevance.

To take into account multiple resources at the same time, we use the Equation 3.7, a weighted average for all resources' *Capacity Class* percentiles to calculate the final probability in determining the value of *forgetme* field.

$$Probability = \left(1 - \sum_{r=1}^{T} p_r \times w_r\right) \times P \text{ such that } \sum_{r=1}^{T} w_r = 1 \tag{3.7}$$

T **(Number of resources)** The total number of resources considered when calculating the probability for the *forgetme* field.

w_r **(Weight of Resource r)** A positive real number between 0 and 1 that represents the weight of contribution for the resource r into the probability for the *forgetme* field.

From Equations 3.5, 3.6 and 3.7, if a node n estimates that the majority of the network is more powerful than itself, n will tend to send messages with the *forgetme* field as true, asking nodes to remove n's contact information from their routing tables. This decreases the likelihood of messages reaching n, reducing n's workload. If a node q is a more powerful relative to their estimation of capacity of the network, q will tend to send messages to nodes with the *forgetme* field as false, increasing q's workload.

3.2.7 Changing K in K-buckets

To increase the workload of powerful nodes, we decided to manipulate maximum size of a node's *k-buckets*. The new K parameter for the k-buckets is calculated every time a new ResEst random walk finishes and will not replace the Kademlia's k parameter, we will exclusively only change the maximum size of a node's *k-buckets* in its routing table. The Equation 3.8 will be used to determine the new maximum size for the *k-buckets*.

$$K = k + \begin{cases} \lfloor (2p_r - 1) \times PC \rceil & , p_r > 0.5 \\ \\ 0 & , p_r \leq 0.5 \end{cases} \tag{3.8}$$

K The new maximum size for *k-buckets*.

k The original Kademlia k parameter value.

PC An positive integer that represents the maximum increase to the original ***k*** parameter. The maximum size for a *k-bucket* will be $K = k + PC$.

Whenever ***K*** changes for node n, n will iterate through all of its *k-buckets*. if K is greater than before, n will keep adding nodes' contact information from the replacement cache to their respective *k-bucket* until the k-bucket size is K or the replacement cache is empty. If K is lower than before, n will remove random nodes' contact information from the *k-bucket* and its respective replacement cache until both sizes are K or lower.

When a node Q increases its K value, it will send messages to a larger variety of nodes, increasing the likelihood of more nodes in the network knowing its contact information, which increases the chance of messages reaching Q, increasing Q's workload.

4

Evaluation and Analysis

4.1 Simulated Experiments

4.1.1 Experimental Environment

Our simulated experiments were performed using PeerSim [26], a P2P system simulator. Both our solution and and the original Implementation of Libp2p's Kademlia have been adapted to work under PeerSim. In this simulator, there is only one thread that executes everything, dealing with every event (for example: timers, message delivery, control events, etc...) and or message, which means no concurrency throughout the whole simulation. In the simulation, time is simulated by dealing with every event that is supposed to happen at the current time. After every event is dealt with, time moves by one unit. Our unit of time used was milliseconds, which means nothing that happens in the simulation will take, for example, half of a millisecond. If the current simulated time is 1000, only events scheduled to happen at 1 second after the start of the simulation will be processed. Every latency between any node to any node is fixed at 100 ms.

Every node has a simulated simplified CPU. Every node has one processing core (meaning only one event gets processed at a time), and a queue of to be processed events. Each event takes the same amount of time to process, regardless of its contents. The processing time can change from node to node, and processing time may change for a single node throughout the simulation.

Besides a simulated simplified CPU, nodes also have have a simulated simplified bandwidth. Every event that is supposed to use bandwidth , like messages between nodes, has a fixed bandwidth cost. Every node has 2 different number of *tokens*, each representing available upload and download bandwidth. When a message is to be received, if the message's cost *"tokens"* is lower or equal to the node's download tokens available and the download queue is empty, it decreases the node's download tokens by its cost and is delivered to the CPU. Otherwise, the message is appended to the download queue. The same procedure happens when sending a message. If a node wants to send a message and the message's cost is lower or equal to the node's upload tokens, the message is sent.

Otherwise the message is appended to the upload queue. Events like timers have 0 cost, ignore the bandwidth procedure and go directly to the CPU. Every second, every node regains its upload and download tokens and goes through the send queue first to process to be sent messages, and then the receiving queue to process incoming messages.

A node's processing time for each event and how many download/upload tokens is allowed to have is defined statically in a file. This ensures any node will always know its initial processing time and bandwidth tokens.

4.1.2 Experimental Settings

The workload created in order to evaluate our experiments involves every node performing one find operation at a time, starting a new one whenever the current find operation finishes. Before the workload starts, all nodes have execute the find operations to join the network and the store operations for all the keys and values that will be queried during the workload. Each experiment will last 25 hours in simulated time.

For hardware, since these experiments are performed in a simulator and time is also simulated in the experiments, hardware choices for performing these experiments are irrelevant as it will not change the outcome or final results of the experiments.

To simulate resource reservation of a node's resources for other purposes beyond the primary workload, we devised a schedule for every node that dictates at what hour in the experiment will each node's reservation start, losing some capacity of its resources. Every reservations starts randomly between the specified hour or 30 minutes after the specified hour.

Whenever the reservation starts, the node will lose some capacity for each resource. In our case, bandwidth tokens will be decreased and processing time for each event will be increased by a flat amount, regardless of the initial values for the resources. When the reservation ends, the node will have its resource capacity reset as it was before the reservation. Each reservation will last for 14 hours in simulated time and any reservation can start from the first hour of the experiment to the last hour.

During the reservation of a node, 50 events will be sent to the node to simulated activity that does not contribute to the primary workload of the experiment. These events will be sent to the node between 10 to 100 millisecond intervals to simulate activity in the reserved resources. These reservation *"activity"* events will stop being sent whenever the reservation for a node ends.

4.1.2.1 Parameters

In our simulated experiments, we devised 3 different network configurations: Balanced, Weak and a Strong configuration.

In all of these configurations, we can assure the following properties:

- Every message from every protocol will have a bandwidth cost of 200 tokens.
- Every timer or event that does not need to use bandwidth will have a cost of 0 tokens.
- The maximum amount time each node can take to process an event is 50 ms.
- The minimum amount time each node can take to process an event is 1 ms.
- The minimum amount of tokens each node will have for both download and upload is a 2000.
- The maximum amount of tokens each node will have for both download and upload is a 18000.
- The distribution of starting hours for the reservations is uniform for all hours.
- The distribution of processing time and bandwidth tokens is Normal with the mean and standard deviation depending on the configuration.

In our experiments, we used 3 different network configurations and distributions of capacity: Balanced, Strong and Weak. All configurations use Normal Distribution to generate the initial values of processing time and bandwidth tokens for every node. For processing time, all configurations have a standard deviation of 12.25 ms. For bandwidth tokens, all configurations have a standard deviation of 4000 tokens.

For the Balanced configuration, the processing time mean was 25.5 ms and the bandwidth tokens mean was 10 000 tokens. For the Strong configuration, the processing time mean was 13.25 and the bandwidth tokens mean was 14 000. And finally, for the Weak configuration, the processing time mean was 37.75 and the bandwidth token mean was 6000.

For Kademlia's configurable parameters, we chose to have a different k and α since the quantity of nodes in the network is very small (1000 nodes). k will have a value of 8 and α a value of 4. The rest of the configurable parameters are their default values from Libp2p's Kademlia.

For HyParView's configurable parameters, we used the protocol equation to determine the Active View and Passive View maximum size, 5 for the Active View and 30 for the Passive View. The rest of the configurable parameters are their default values.

For ResEst's configurable parameters, we used a max margin of error of 0.15 and confidence coefficient of 95% since this combination has the less histogram error in HyParView [4].

For our solution's configurable parameters, we chose the following:

- **N** (Capacity Classes or Max Capacity Class): The total number of *Capacity Classes* chosen was 10.
- **S** (Curve Steepness): The curve steepness chosen was 7.2.
- **R** (Resource capacity): For this parameter, we chose the CPU and bandwidth for the resources to estimate node capacity. The aspect of both resources that we chose to represent its capacity is their usage(or its availability). The Resource Capacity value chosen was 1, while their usages were a value between 0 and 1.
- *Weight of the Resource* ***r***: For both CPU and bandwidth, we chose the same weight of contribution: 0.5.
- **P** (Max Probability): The maximum probability chosen was 0.5.
- (Max Sliding Window size for ResEst samples): The maximum size for the sliding window filled with ResEst random walks was 10.
- w_h (Weight of relevance for the most recent random walk): The weight chosen for the most recent random walk was 0.7.
- w_l (Weight of relevance for the other random walks): The weight chosen for the rest of the random walks was 0.3.
- **PC**: The maximum increase for the size of *k-buckets* we chose was 12.

4.1.3 Experimental Metrics

In order to determine if our solution is working as intended, we need metrics to able to compare both experiments using our solution and the original implementation without using our solution. The following metrics were chosen to evaluate our simulated experiments:

- **Completed Find Operations**: The total number of completed find operations will help determine whether or not our solution does improve throughput compared to the original implementation. More completed Find Operations in the same amount of time points to increase in performance if the success rate of those completed find operations stays roughly the same.
- **Messages Received according to *Capacity Class***: The average messages received for nodes with a specific average *Capacity Class* can help us verify if our strategy is working as intended. According to our strategy, lower *Capacity Class* nodes should receive less messages, which reduces the amount of work they have to perform. On the other hand, our strategy should also make higher *Capacity Class* nodes receive more messages, which increases the amount of work they have to perform.

- **In-Degree according to *Capacity Class***: The average in-degree for nodes with specific average *Capacity Class* can help verify that our is working as intended. Higher average *Capacity Class* nodes should have a higher in-degree according than nodes with a lower average *Capacity Class*.

4.1.4 Experimental Results

We first present the results in all described configurations in light of the following metrics: In-degree and Messages Received according to average Capacity Class.

From the strategy and solution described in Chapter 3, we should expect for higher average *Capacity Class* nodes to have a higher in-degree than lower average *Capacity Class* nodes. We should also expect lower average *Capacity Class* nodes to receive less messages than higher average *Capacity Class* nodes.

From Figures 4.1 to 4.6, each *Capacity Class* in the graphs represents all nodes that have the same *Capacity Class* calculated using Equation 4.1.

$$NodeCapacityClass = \frac{SumOfAllResourceCapacityClasses}{SumOfAllResourceWeightsOfContribution} \tag{4.1}$$

We will present the graphs for the average messages received and in-degree for each average *Capacity Class* in a specified configuration, for both implementations. Each graph represents the average of 6 experiments, 3 for each implementation. The collected information was performed in the last 30 minutes of each set of experiments for a given configuration.

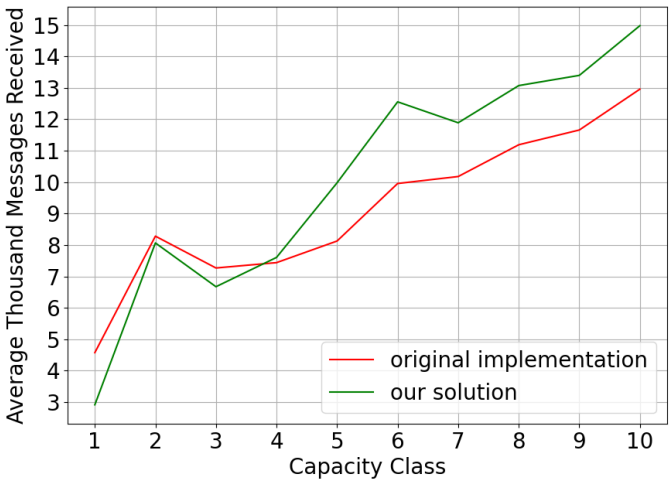

Figure 4.1: Average Messages Received for each bandwidth *Capacity Class* in the Balanced configuration for both implementations.

In Figure 4.1, we can observe that that our solution follows the same trend as in the

original implementation. However, our solution, compared to the original implementation, reduced the amount messages received for lower *Capacity Class* nodes and increased that amount for higher *Capacity Class* nodes. This behavior is in line with our strategy of reducing workload by reducing the amount of messages received and increasing workload by increasing the amount of messages received.

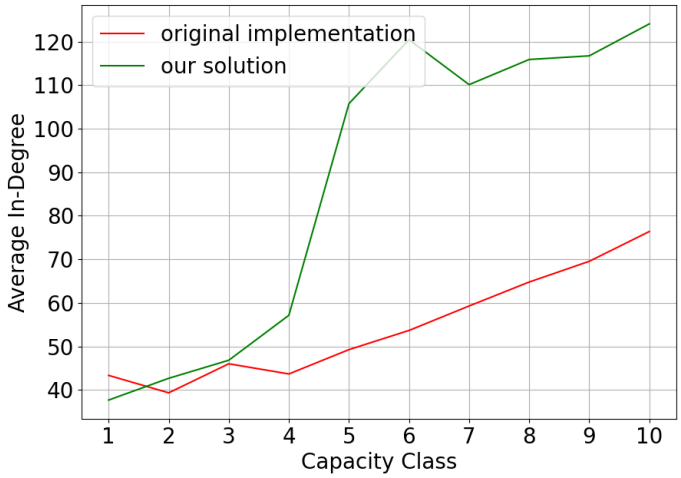

Figure 4.2: Average In-Degree for each bandwidth *Capacity Class* in the Balanced configuration for both implementations.

In Figure 4.2, we can observe that higher *Capacity Class* nodes have a much higher in-degree than lower *Capacity Class* nodes, a pattern that the original implementation does seem to show although in a much smaller scale. This is an expected behavior from our solution. High capacity nodes need to have a higher in-degree in order to increase the likelihood of messages reaching those nodes, increasing their workload. For low capacity nodes, a lower in-degree is needed in order to decrease the likelihood of messages reaching these nodes, decreasing their workload.

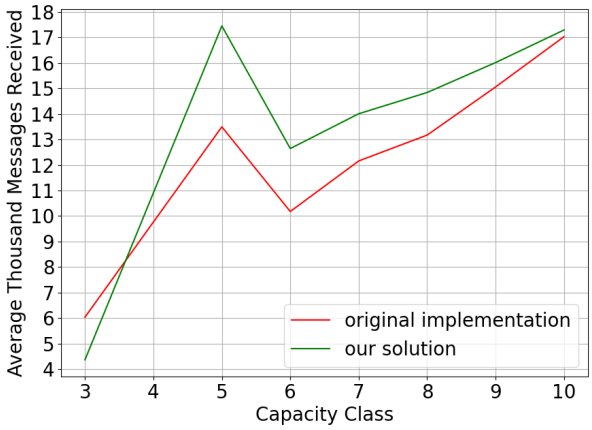

Figure 4.3: Average Messages Received for bandwidth *Capacity Class* in Strong configuration for both implementations.

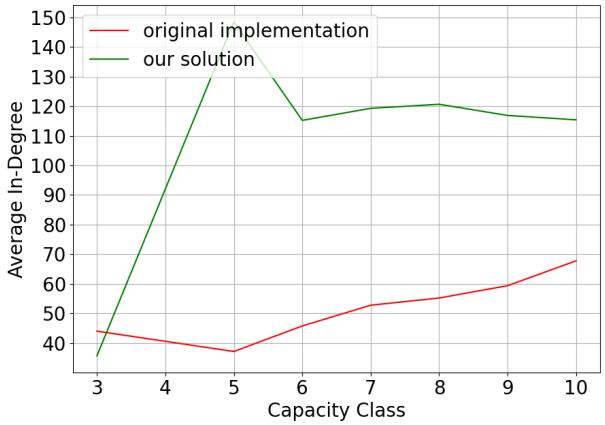

Figure 4.4: Average In-Degree for bandwidth *Capacity Class* in the Strong configuration for both implementations.

Figures 4.3 and 4.4 present the same type of graphs as Figures 4.1 and 4.2, using the Strong network configuration instead of the Balanced network configuration. In Figure 4.3, we see the same thing happening in Figure 4.1. Both implementations follow the same pattern and our solution slightly deviates from the original implementation by reducing received messages for lower capacity nodes and increasing received messages for more powerful nodes. The peak in messages received we see for nodes with a *Capacity Class* of 5 might be due to excess capacity or lack of a more resource heavy workload to differentiate nodes with *Capacity Classes* from 5 to 10. Another reason for this peak could be the uneven distribution of capacity created much more nodes with a *Capacity Class* of 5 compared to nodes with other *Capacity Classes*.

From Figure 4.4, we can see a rough plateau in variation of in-degree for nodes past the *Capacity Class* of 5. This pattern can be due to the increase of messages received for nodes with a *Capacity Class* of 5, which we can confirm in the previously presented Figure 4.3. Besides that plateau pattern, our strategy decreased the in-degree for lower capacity nodes and increased the in-degree for higher capacity nodes, just as expected.

And finally, Figures 4.5 and 4.6, present the same type of graphs as the previous Figures for the Weak network configuration.

We can observe in Figure 4.5, a similar situation presented as with Figure 4.1. Nodes with a higher *Capacity Class* received more messages than nodes with lower *Capacity Classes* when using our solution. The original implementation also follows the same trend but to a lesser scale.

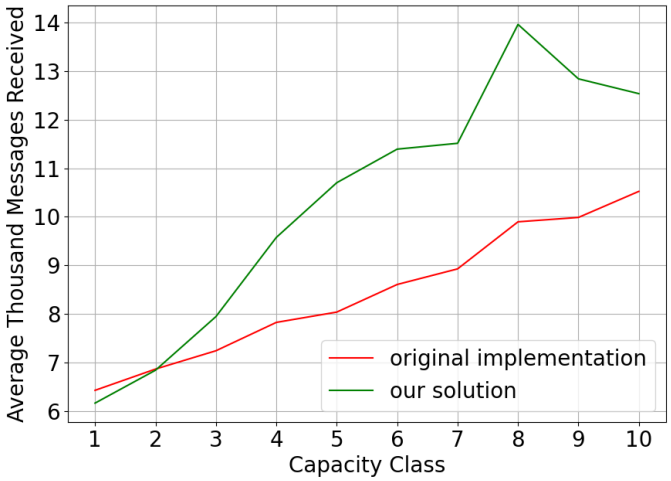

Figure 4.5: Average Messages Received for each bandwidth *Capacity Class* in Weak configuration for both implementations.

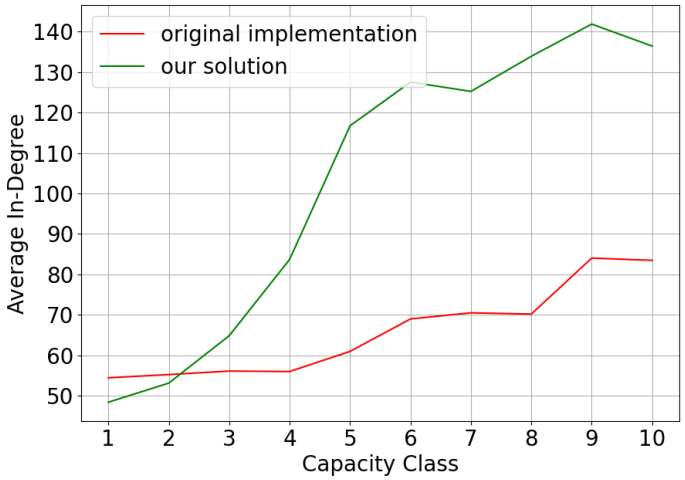

Figure 4.6: Average In-Degree for each bandwidth *Capacity Class* in Weak configuration for both implementations.

In Figure 4.6, when using our solution, nodes with higher capacity have a significantly higher in-degree that nodes with lower capacity. Our solution has differentiated the in-degree according to node capacity effectively. On the other hand, the original implementation has differentiated in-degree according to node capacity in a very small scale.

In summary, even though the original implementation already tends to differentiate messages received and in-degree according to node capacity, our solution emphasizes and boosts that differentiation to a much larger effect in order to effectively distribute workload according to node capacity.

4.1.4.1 Results: Completed Find Operations

Finally, the following graphs presented will compare our solution and the original implementation according to Completed Find Operations. The success rate of these operations

for all experiments was roughly the same. The worst success rate of all combinations of configuration and implementation was 99.498% and the best success rate was 100%.

From as described in Chapter 3, we should expect our solution to increase the completed find operations compared to using the original implementation, which leads to an increase in overall performance.

The Figures 4.7, 4.8 and 4.9 are bar graphs of the average total Find Operations performed by each implementation in a specified configuration. Each graph represents 6 different experiments, 3 per implementation. In these graphs, there is also a margin of error for each implementation, with a confidence level of 95%.

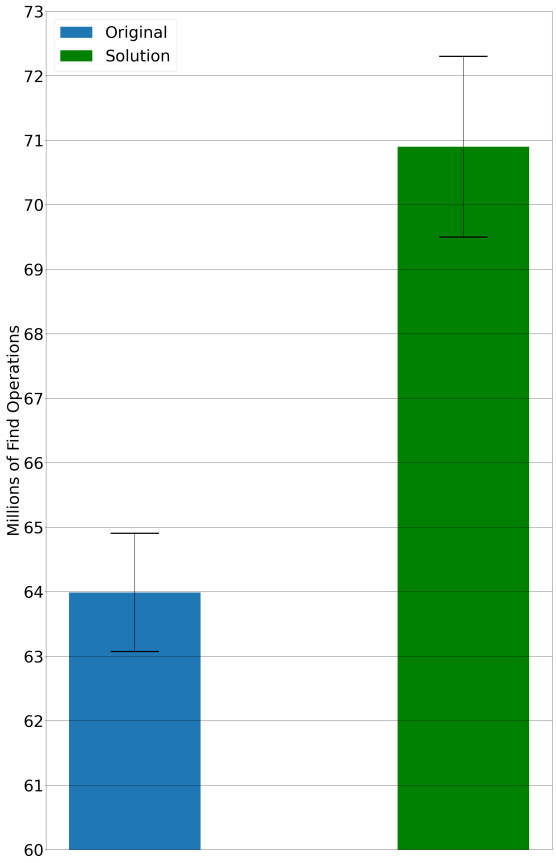

Figure 4.7: A bar graph for Total Completed Find Operations and the margin of error for both implementations in a Balanced configuration.

In Figure 4.7, we can observe that our solution has indeed increased the total completed find operations in a Balanced network configuration. It has performed on average 70.9 million find operations with a margin error of 1.402 million find operations. Moreover, the original implementation has a mean of 63.989 million operations and a margin error of 0.917 million operations. With this configuration, our solution has on average has completed 6.911 million (10.8%) more find operations compared to the original implementation.

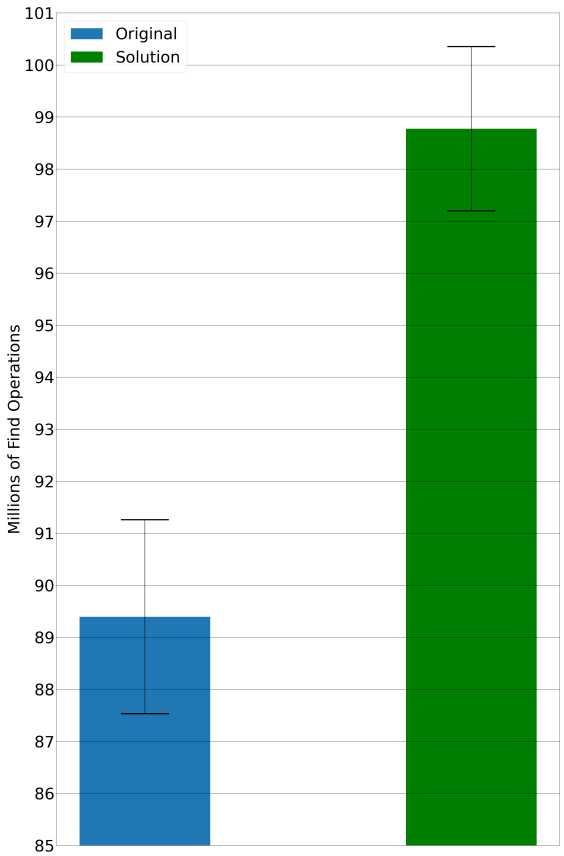

Figure 4.8: A bar graph for Total Completed Find Operations and the margin of error for both implementations in a Strong configuration.

In Figure 4.8, the difference in completed find operations between our solution and the original implementation with a Strong network configuration is slightly lower than with a Balanced network configuration. Our solution has performed on average 98.772 million operations with a margin of error of 1.58 million operations. The original implementation performed on average 89.394 million operations with a margin of error of 1.864 million operations. With this configuration, our solution has on average performed 9.378 million (10.491%) more operations relative to the original implementation.

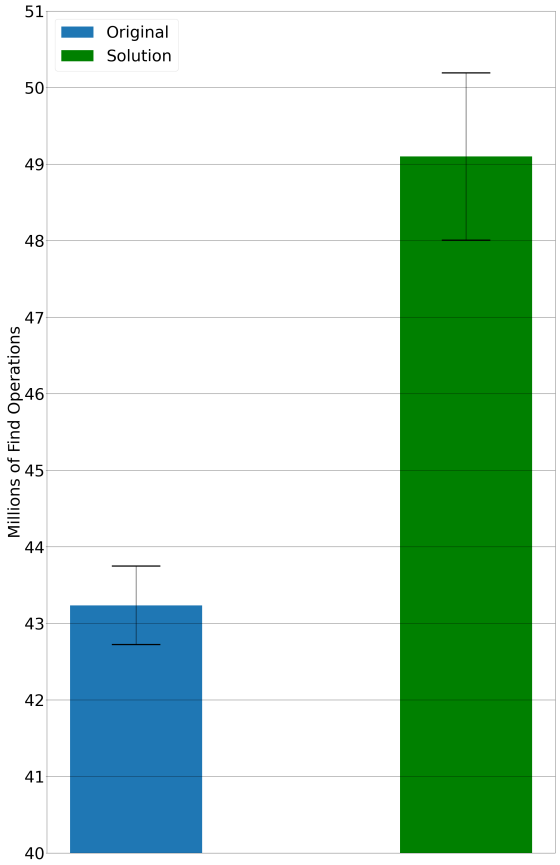

Figure 4.9: A bar graph for Total Completed Find Operations and the margin of error for both implementations in a Weak configuration.

And finally, in Figure 4.9, we can also observe that our solution has performed on average 49.099 million operations with a margin of error of 1.093 million operations. The original implementations performed with a mean of 43.235 million operations with a margin of error of 0.513 million operations. Our solution performed on average 5.864 million (13.563%) more operations than the original implementation.

Overall, our simulated experiments have proven that our strategy can work in a simulated environment in our devised configurations and has potential to work in real-life scenarios.

4.2 Emulated Experiments

4.2.1 Experimental Environment

Our emulated experiments were performed in Docker virtual networks, with each node having a container by itself, with one dedicated CPU core[1]. An allowance was applied to every node for how much of the delegated core can the node use. The sum of node

[1]This includes *"logic"* cores derived from hyper threading processors.

allowances in all experiments for the same core will never be above 100%. In all experiments, 2 CPU cores were spared from being delegated as to not overload the host machine and prevent the operating system to fight for resources against the node containers.

We implemented our solution and the original Kademlia protocol in Java over the Babel[1] network. This framework is useful for the development of distributed protocols and systems, since it simplifies network management, provides the ability send and receive to notifications between protocols, set timers and provides a simple way to send messages between different processes (in our case containers/nodes).

Every established connection is a TCP connection and Kademlia and HyParView do not share connections. If node Q has connection in HyParView with node N, node Q does not need to have a connection with node N in Kademlia and vice versa. We implemented our solution this way so that HyParView works independently from Kademlia, providing ResEst a membership to estimate the capacity in the network. Keeping connections separate helps to avoid disturbances and unexpected situations in both protocols.

4.2.2 Experimental Settings

The workload for the emulated experiments is the same as the workload in the simulated experiments. Before this workload, all nodes need to already have executed the join operations and store operations for all the keys and values that will be queried during the workload. Each experiment lasted 8 hours.

For hardware, we used a host machine with a AMD EPYC 7281 CPU with 16 cores and 32 threads, and 128 GiB DDR4 2666 MHz for RAM. All experiments were ran with the same configurable protocol parameters in a network of 100 nodes with the same hardware. Every node had an allowance of 1 Mbps of upload/download, to avoid overusing the host machine's bandwidth (1 Gbps of upload and download). This allowance was enough that no single node never overused its bandwidth.

Since all nodes will run on the same machine, calculating resource usage is tricky and can be unreliable. In order for every node to assess their resource usage, we implemented a workload program that will run inside each node's container, alongside the Babel node Java application.

This program will create a CPU workload based the target resource usage to emulate. Whenever a node will attempt to assess its resource usage, a static number will be returned. That same number will be used to configure the workload in the workload generator program. Whenever that static number switches to new number, both programs will reflect that change simultaneously. For example, if a node assesses its current CPU usage is 100%, the workload program will create a 100% workload, using the node's resources to *"emulate"* resource usage.

[1] https://github.com/pfouto/babel-core

4.2.2.1 Parameters

In our emulated experiments, we have two types of nodes: Weak CPU and Strong CPU nodes. Weak CPU nodes have an allowance of 10% of the delegated CPU core and Strong CPU nodes have an allowance of 30%. The network configurations used in our emulation experiments have the same names as the network configurations from the simulated experiments but with different settings. Furthermore, we created two types of CPU workload distributions to emulate CPU usage: Balanced workload and Biased workload.

For the Balanced configuration, one half of the nodes in the network are Strong CPU nodes and the other half Weak CPU nodes. For the Weak configuration, 80 nodes are Weak CPU nodes and 20 nodes are Strong CPU nodes. And finally, for the Strong configuration, we have the reverse situation of the Weak configuration. 20 nodes in the network are Weak CPU nodes and the other 80 nodes are Strong CPU nodes.

Every CPU workload cycles through an active and an inactive state, emulating 100% and 0% CPU usage respectively. Each hour of all experiments, all active workloads will be turned into inactive workloads and vice-versa.

For all network configurations, the Balanced CPU workload distribution has half of the nodes for each type of CPU start with the CPU workload in an inactive state and the rest of nodes in an active state.

The Biased CPU workload profile will depend on the network configuration. Whenever a Balanced network configuration is used, the Biased CPU workload will have 10 nodes for each type of CPU node to have the workload in an active state and 40 nodes in an inactive state. On the other hand, whenever a Weak or Strong network configuration is used, the type of CPU node that is in the minority will have the workload in an active state and the rest of the nodes in an inactive state. For example, if we are using the Weak configuration, only 20 nodes will start the workload with an active state while the other 80 nodes will start with an inactive state. If we are using the Strong configuration, the reverse is true. 80 nodes that are Strong CPU nodes will start the workload in an inactive state and the Weak CPU nodes will start in an active state. Since we have 2 states and an even amount of hours in all experiments, both workloads will be ran the same amount of hours in all experiments and nodes.

For Kademlia's configurable parameters, we chose to have a different k and α since the quantity of nodes in the network is very small (100 nodes). k will have a value of 8 and α a value of 4. The rest of the configurable parameters are their default values from Libp2p's Kademlia.

For HyParView's configurable parameters, we used the protocol equation to determine the Active View and Passive View maximum size, 4 for the Active View and 20 for the Passive View. The rest of the configurable parameters are their default values.

For ResEst's configurable parameters, we used a max margin of error of 0.15 and confidence coefficient of 95% since this combination has the less histogram error in HyParView [4].

For our solution's configurable parameters, we chose the following:

- **N** (Capacity Classes or Max Capacity Class): The total number of *Capacity Classes* chosen was 10.
- **S** (Curve Steepness): The curve steepness chosen was 7.2.
- **R** (Resource capacity): For this parameter, we chose the only the CPU for the resource to estimate node capacity. The aspect of the resource that we chose to represent its capacity is its usage(or its availability). The Resource Capacity value chosen was 1, while the CPU usage was a value between 0 and 1.
- **P** (Max Probability): The maximum probability chosen was 0.5.
- (Max Sliding Window size for ResEst samples): The maximum size for the sliding window filled with ResEst random walks was 10.
- w_h (Weight of relevance for the most recent random walk): The weight chosen for the most recent random walk was 0.7.
- w_l (Weight of relevance for the other random walks): The weight chosen for the rest of the random walks was 0.3.
- **PC**: The maximum increase for the size of *k-buckets* we chose was 12.

4.2.3 Experimental Metrics

In order to determine if our solution is working as intended in our emulated experiments, we need metrics to able to compare both our solution and the original implementation. The following metrics are the ones used for our emulated experiments:

- **Completed Find Operations**: The total number of completed find operations will help determine whether or not our solution does increase find operation throughput compared to the original implementation, improving overall performance. A higher throughput translates to improved performance if the success rate of those completed find operations is roughly the same.
- **Average Messages Received per type of Node CPU**: The average messages received for a specific type of node CPU can tell us how much work a specific type of node had. Our solution attempts distribute more work towards Strong CPU nodes and less work towards the Weak CPU nodes.

4.2.4 Experimental Results

We first present the results in light of total messages received for each of the described configurations and CPU workload types. For each combination of configuration and CPU workload type, we performed 3 different experiments with the same settings and parameters.

From Figures 4.10 to 4.12, we present the average messages received for each type CPU in all network configurations, CPU workloads and implementations. In each graph, each bar represents the average number of messages received of nodes with a specific type of CPU, in that particular implementation, network configuration and CPU workload. In other words, each experiment we extract the total number of messages received for all nodes with a specific type of CPU and divide it by the number of nodes with that type of CPU in the experiment. For 3 experiments, we average that value and represent it by a bar and a margin of error in those Figures. In these graphs, the margin of error for each implementation has a confidence level of 95%.

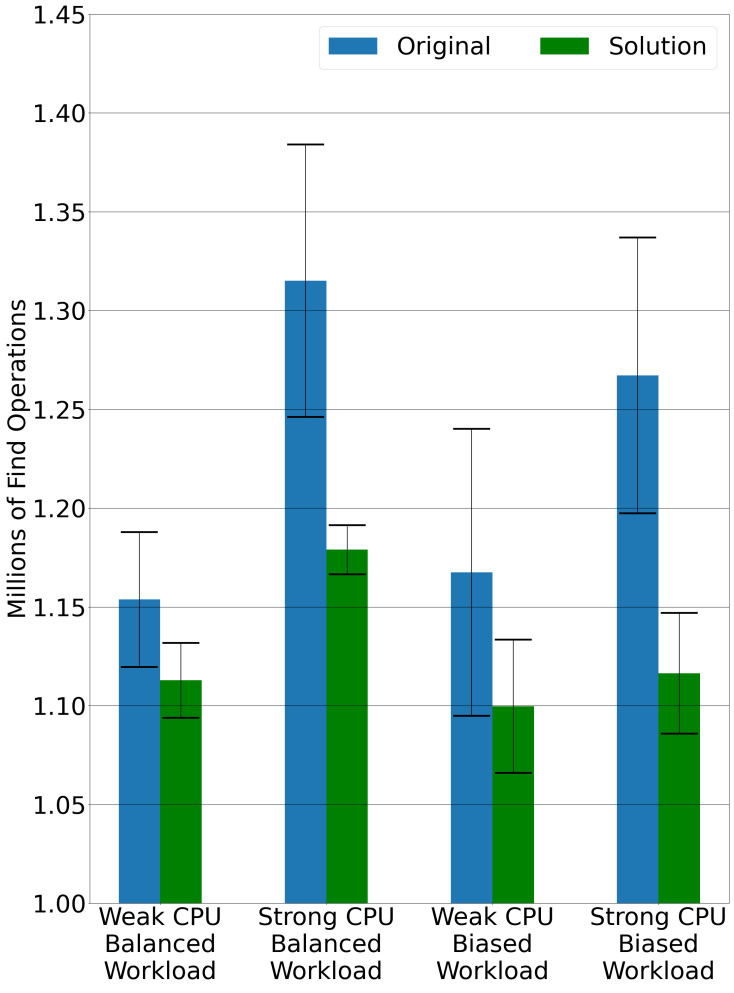

Figure 4.10: A graph for the average messages received for a type of node CPU, in a implementation with a Balanced configuration.

In Figure 4.10, when it comes to margin of error overall, our solution for each type of CPU node and in both CPU workloads, has a much smaller margin of error and reduced number of messages received when comparing to the original implementation. This can be attributed to the *forgetme* field in messages making nodes forget each other, decreasing the amount of messages sent overall in the system.

Furthermore, we were able to reduce the difference in messages received when it comes to experiments with the Biased CPU workload. By using our solution, we were able to distribute work more evenly, depending on how many nodes have a busy the CPU at a particular time, sometimes independently of the type of CPU.

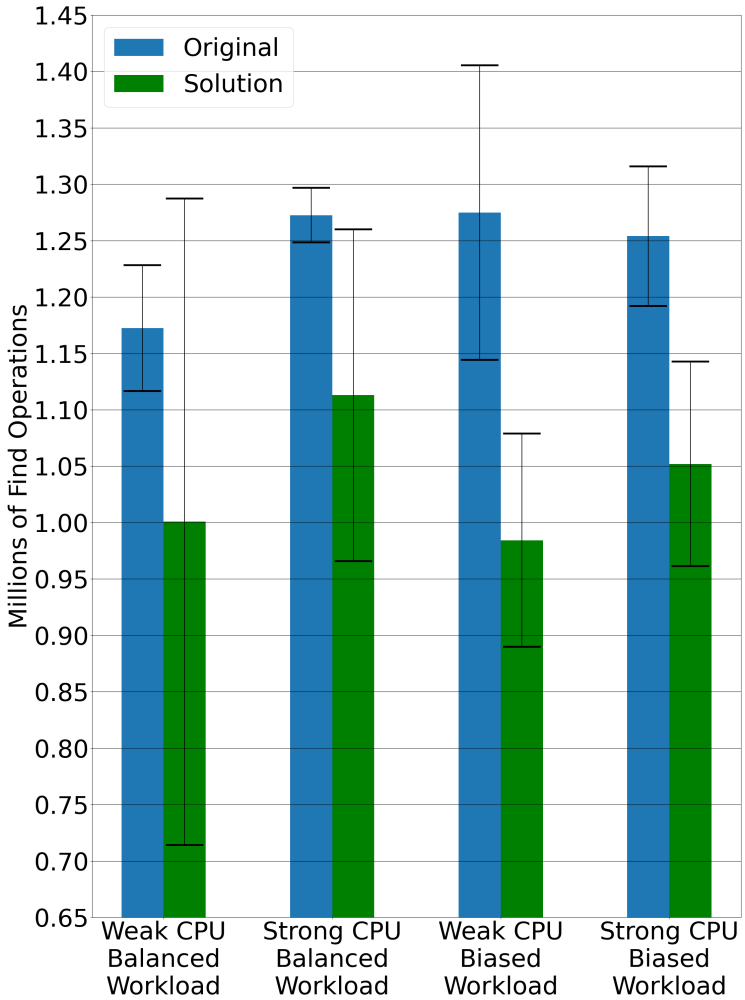

Figure 4.11: A graph for the average messages received for a type of node CPU, for both implementations with a Strong configuration.

In Figure 4.11, unfortunately, we see the opposite of what happened in Figure 4.10. The margin of error for the Balanced workload CPU for our solution is significantly larger compared to the original implementation.

This can be due to lack of convergence detection in our solution. Our solution, no matter the current distribution of work, may attempt to *"over-correct"* beyond the current

distribution of work, regardless if it is optimal. This could be due to our reliance on random probability when changing the in-degree of nodes. Our solution may reduce in-degree just the way we want but it can also not reduce enough or reduce too much.

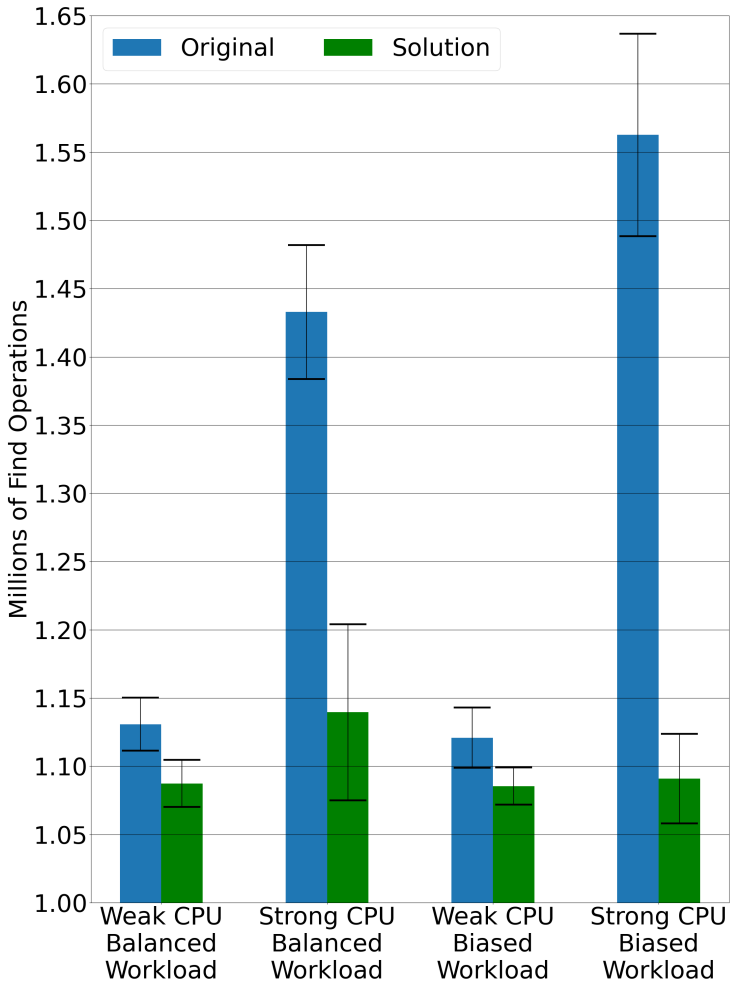

Figure 4.12: A graph for the average messages received for a type of node CPU, for both implementations with a Weak configuration.

Finally, in Figure 4.12, the number of messages received for Strong CPU nodes was massively reduced with our solution. This massive reduction may be attributed the *forgetme* field reducing the in-degree of Weak CPU nodes, which can massively reduce the amount of redundant messages. Furthermore, we again see a reduction of the margin of error in both CPU workloads for Strong CPU nodes. These results can be due to a more fair and consistent distribution of work that our solution attempts to provide.

4.2.4.1 Results: Completed Find Operations

Finally, we will present the graph results in light of the Completed Find Operations for all of the described network configurations, CPU workloads and implementations. The success rate of these operations for all experiments was roughly the same. The worst

success rate of all experiments was 99.53% and the best success rate was 100%.

From the strategy and solution described in Chapter 3, we should expect the completed find operations on the experiments using our strategy to be higher than the experiments using the original implementation.

The Figures 4.13, 4.14 and 4.15 are bar graphs of the average total completed Find Operations performed by each implementation in a specified configuration and CPU workload. Each bar represents 3 different experiments, 3 per combination of CPU Workload and implementation. In these graphs, the margins of error have a confidence level of 95%.

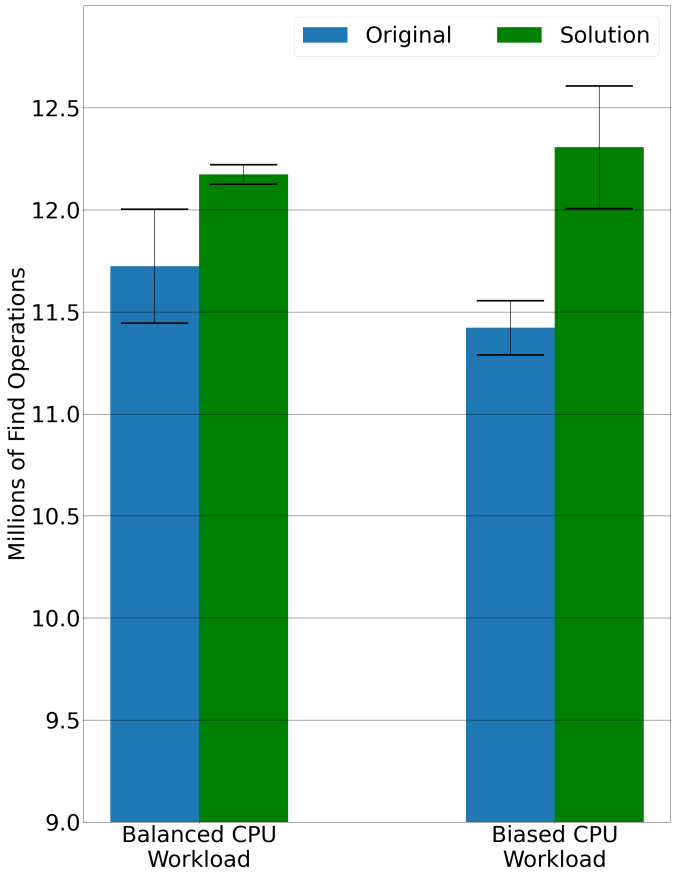

Figure 4.13: A graph for the completed find operations received for both implementations and CPU workloads, with a Balanced network configuration.

In Figure 4.13, our solution performed more find operations than the original implementation, in both CPU workloads.

Our solution has completed on average 12.173 million find operations for the Balanced CPU workload with a margin of error of 47 489 find operations. For the Biased CPU workload, our solution performed on average 12.306 million find operations with a margin of error of 300 470 find operations.

For the original implementation, it completed on average 11.724 million find operations with a margin of error of 278 896 find operations for the Balanced CPU workload and, for the Biased CPU workload, an average of 11.422 million find operations with a margin of error of 132 996 find operations.

Furthermore, with the Balanced CPU workload, our solution has performed 449 thousand (3.830%) more operations than the original implementation. For the Biased CPU workload, 884 thousand (7.739%) more operations were completed than the original implementation.

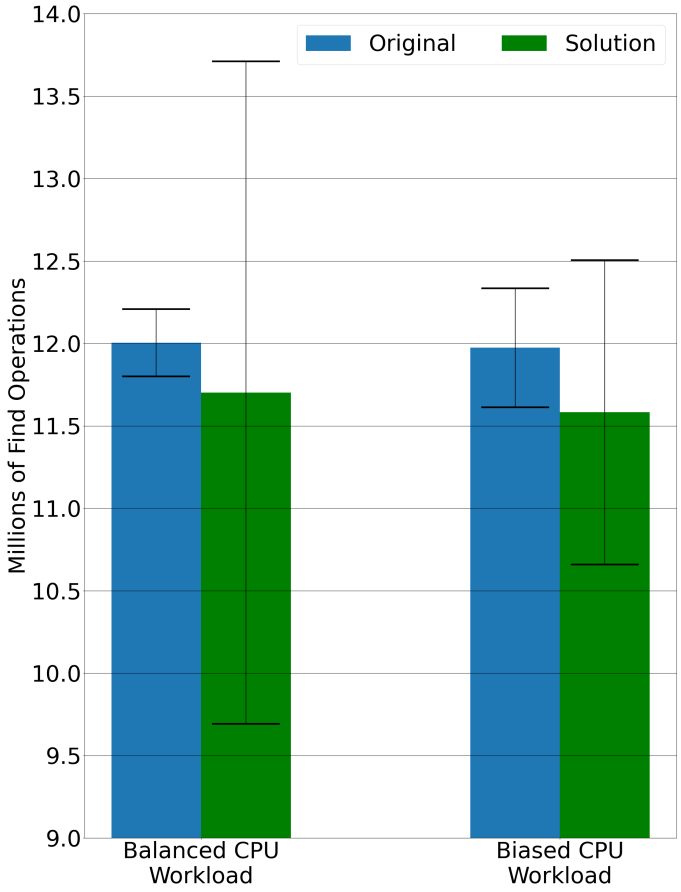

Figure 4.14: A graph for the completed find operations received for both implementations and CPU workloads, with a Strong configuration.

For the Strong configuration we have Figure 4.14. Our solution has on average performed less than the original implementation in both CPU workloads.

On average, our solution performed 11.702 million operations with a margin of error of 2.009 million operations for the Balanced CPU workload. For the Biased CPU workload, it performed on average 11.582 million operations with a margin of error of 923 524 operations.

The original implementation performed on average 12.004 million operations with a margin of error of 204 430 operations for the Balanced CPU workload and 11.973 million

operations with a margin of error of 360 908 operations for the Biased CPU workload.

Moreover, the original implementation completed on average more Find Operations than our solution, performing on average 302 thousand (2.581%) more operations with the Balanced CPU workload and 391 thousand (3.376%) more operations with the Biased CPU workload.

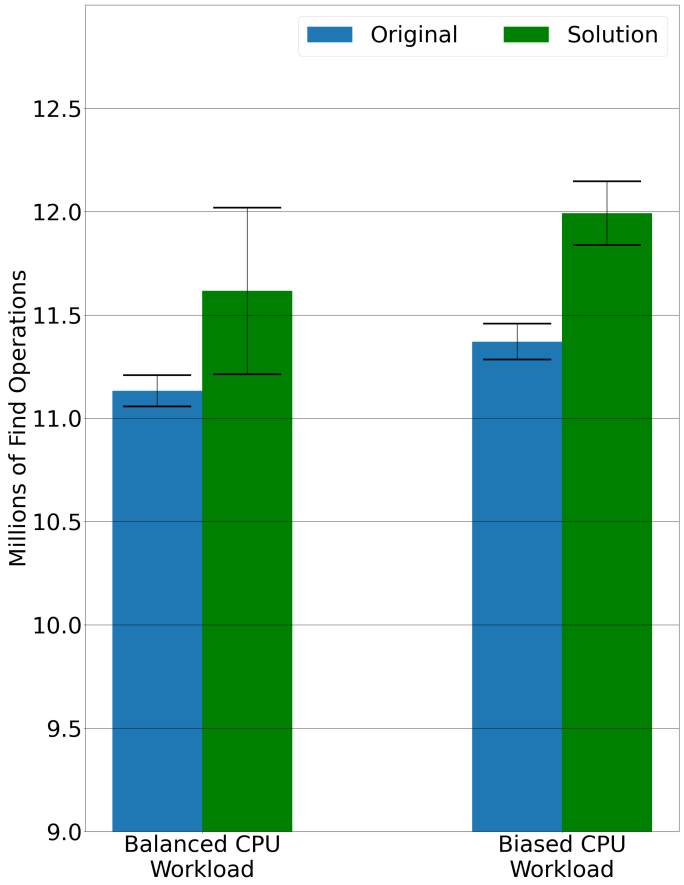

Figure 4.15: A graph for the completed find operations received for both implementations and CPU workloads, with a Weak configuration.

And finally, for the Weak network configuration we have Figure 4.15. Our solution has a larger margin of error compared to the original implementation. However, our solution outperformed the original implementation.

Our solution managed to complete on average 11.616 million operations with a margin of error of 403 312 operations with the Balanced CPU workload. For the Biased CPU workload, it performed on average 11.992 million operations with a margin of error of 154 446 operations.

Furthermore, the original implementation has on average completed 11.133 million operations with a margin of error of 76 216 operations with the Balanced CPU workload. On the other hand, with the Biased CPU workload, the original implementation managed to complete on average 11.371 million operations with a margin of error of 86 818

operations.

4.3 Summary

Overall, our solution has shown that we are on the right track when it comes to fairly distributing work according to node capacity. Our simulation results in Figures from 4.1 to 4.9 show that our strategy of changing in-degree and messages received according to a node's capacity in order to increase Find Operation throughput is working as expected.

However, our solution is far from perfect in tackling the problem presented in this thesis. In Figure 4.14, the overall performance was not only not increased but in fact decreased substantially compared to the original implementation.

In summary, our solution does have potential to become the base for a useful and interesting strategy to distribute work more fairly in Structured Overlays. However, we are aware that our testing environments are not as representative of real world scenarios as we would have liked. For example, in our emulated environment, we had to rely on emulating CPU usage in an artificial cyclical way due to the trickiness and unreliability of monitoring owned resources in a Docker Container environment in the same host machine.

And finally, we do believe that our results have revealed insightful information towards devising a load balancing strategy, capable of providing a solution to the problem presented in this thesis.

5

Conclusion and Future Work

In the previous chapters we have present and discussed the challenges in this thesis we wanted to address, discussed the related work, proposed our solution, and finally, evaluated and analyzed our simulated and emulated experiments, in order to compare our solution with the original implementation without our solution. In this chapter, we conclude this thesis. Furthermore, this chapter is divided into two sections: in Section 5.1 we will summarize the work performed in this thesis and present our conclusions; in Section 5.2 we end this thesis presenting what could be done to optimize our solution in future works.

5.1 Conclusion

P2P systems are more relevant now more than ever and optimization is something everyone wants to pursue. Load balancing is a topic that is an unexplored topic, specially when it comes to structured overlays. Homogeneity is assumed for all nodes and whatever existing differences between nodes in terms of capacity is sometimes treated as an afterthought, long after a system is already built and running. Node resources might be underused, less powerful nodes might get overloaded, and a potential increase in overall performance may be ignored if load balancing is performed in an unfair manner as we seen in our discussion in Chapter 2.

Though attempts to distribute load more fairly exist, these strategies mostly focus on Unstructured Overlays. The techniques used do not translate well when it comes to Structured Overlays. The goal of this thesis was to create a solution the problem of heterogeneity present in Structured Overlays by devising a load balancing strategy that uses nodes resources more fairly. A strategy that avoids overloading less powerful nodes, increases resource usage for more powerful nodes, and provides an increase in overall performance to the system.

Our solution uses a dynamically calculated probability to affect a node's in-degree in

Kademlia-based system, according to their perceived individual capacity and an estimation of capacity of the network.

We evaluated and analyzed our experiments with our solution, its drawbacks and benefits compared to the Libp2p's Kademlia implementation.

In summary, we developed a load balancing strategy in an attempt to increase overall performance in heterogeneous structured overlays. From our evaluation, our solution did provide, in some of our results, an increase in overall performance and showed signs that we are on the right track towards a useful load balancing strategy.

5.2 Future Work

In the last section of our thesis, we present the following possible future work to better our solution:

Reaching/Detecting Convergence: One of glaring problem of our solution is the workload has been distributed fairly but our solution may attempt *"over correct"* the distribution, beyond its optimal state. Our use of random probability and dynamically calculating the estimation of network capacity may contribute to that problem. It would be interesting to develop a more convergence aware strategy for such a problem in the future.

A more realistic set of experimental environments: In our experimental emulated results, we faced some problems when calculating current CPU usage. We decided to *"emulating"* CPU usage due to the mixed view of Docker Containers hosted on the same machine. Our results may not reflect reality in a trustworthy manner. Due to time constraints, we could not take into account another resource like bandwidth in our experiments. Perhaps creating more realistic set experimental environments with more resources taken into account would reveal how close we are to a useful and robust solution.

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. Leitao. "Topology management for unstructured overlay networks". In: *Technical University of Lisbon* (2012), p. 12 (cit. on pp. 1, 5).

[3] P. Maymounkov and D. Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65 (cit. on pp. 2, 21, 29).

[4] V. H. Menino, P. Á. Costa, and J. Leitao. "ResEst–Algoritmo Distribuıdo para a Inferência de Recursos da Rede". In: (2021) (cit. on pp. 3, 27, 29–32, 34, 41, 52).

[5] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A membership protocol for reliable gossip-based broadcast". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE. 2007, pp. 419–429 (cit. on pp. 3, 6, 27, 29, 30, 32, 34).

[6] *libp2p: A modular network stack*. URL: https://libp2p.io/ (cit. on p. 3).

[7] P. Labs. *Filecoin: A Decentralized Storage Network*. URL: https://filecoin.io/ (cit. on p. 3).

[8] J. Benet. *IPFS - Content Addressed, Versioned, P2P File System*. URL: https://www.ipfs.com/ (cit. on p. 3).

[9] I. Stoica et al. "Chord: a scalable peer-to-peer lookup protocol for internet applications". In: *IEEE/ACM Transactions on networking* 11.1 (2003), pp. 17–32 (cit. on pp. 4, 10, 16–18).

[10] J. Roskind. *QUIC: Multiplexed Transport Over UDP*. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-22 (cit. on p. 5).

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

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

[13] S. Voulgaris, D. Gavidia, and M. Van Steen. "Cyclon: Inexpensive membership management for unstructured p2p overlays". In: *Journal of Network and systems Management* 13 (2005), pp. 197–217 (cit. on pp. 7, 8).

[14] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. "Scamp: Peer-to-peer lightweight membership service for large-scale group communication". In: *Networked Group Communication: Third International COST264 Workshop, NGC 2001 London, UK, November 7–9, 2001 Proceedings 3*. Springer. 2001, pp. 44–55 (cit. on p. 7).

[15] Y. Chawathe et al. "Making gnutella-like p2p systems scalable". In: *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. 2003, pp. 407–418 (cit. on p. 7).

[16] M. F. Kaashoek and D. R. Karger. "Koorde: A simple degree-optimal distributed hash table". In: *Peer-to-Peer Systems II: Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22, 2003. Revised Papers 2*. Springer. 2003, pp. 98–107 (cit. on pp. 8–10, 19).

[17] S. Ratnasamy et al. "A scalable content-addressable network". In: *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. 2001, pp. 161–172 (cit. on pp. 10, 11, 13).

[18] P. Francis. *Yoid: Extending the internet multicast architecture*. 2000 (cit. on p. 11).

[19] A. Rowstron and P. Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001 Proceedings 2*. Springer. 2001, pp. 329–350 (cit. on pp. 13, 14).

[20] A. Rowstron et al. "SCRIBE: The design of a large-scale event notification infrastructure". In: *Networked Group Communication: Third International COST264 Workshop, NGC 2001 London, UK, November 7–9, 2001 Proceedings 3*. Springer. 2001, pp. 30–43 (cit. on p. 16).

[21] A. Montresor, M. Jelasity, and O. Babaoglu. "Chord on demand". In: *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*. IEEE. 2005, pp. 87–94 (cit. on p. 19).

[22] I. Gupta et al. "Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead". In: *Peer-to-Peer Systems II: Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22, 2003. Revised Papers 2*. Springer. 2003, pp. 160–169 (cit. on p. 19).

[23] I. Baumgart and S. Mies. "S/kademlia: A practicable approach towards secure key-based routing". In: *2007 International conference on parallel and distributed systems*. IEEE. 2007, pp. 1–8 (cit. on p. 24).

[24] V. H. Menino. "A Novel Approach to Load Balancing in P2P Overlay Networks for Edge Systems". PhD thesis. NOVA University of Lisbon, 2021 (cit. on p. 26).

[25] V. Venkataraman, K. Yoshida, and P. Francis. "Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast". In: *Proceedings of the 2006 IEEE international conference on network protocols*. IEEE. 2006, pp. 2–11 (cit. on p. 26).

[26] A. Montresor and M. Jelasity. "PeerSim: A Scalable P2P Simulator". In: *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*. Seattle, WA, Sept. 2009, pp. 99–100 (cit. on p. 39).