Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computers

DHT implementation in Java

Bc. Vojtěch Kuzdas

Supervisor: Ing. Peter Macejko
Field of study: Software Engineering
May 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Kuzdas Vojtěch**	Personal ID number:	**485044**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

DHT implementation in Java

Master’s thesis title in Czech:

Implementace DHT v Javě

Guidelines:

Design and implement a library for working with distributed hash tables (DHT) in Java. The library will provide a generic interface for creating, joining, and querying DHT clusters capable of storing and querying data by keys. The library will also allow switching between different DHT algorithms such as Pastry, Kademlia, and Chord, which differ in the way nodes are organized and messages are routed.
In the theoretical part of the thesis, a review of existing DHT algorithms and their properties will be carried out. Furthermore, the gRPC technology that should be used for communication between the nodes of the DHT cluster will be described.
In the practical part of the thesis, a library interface will be designed and implemented to work with the DHT cluster to initialize, connect, disconnect, insert, retrieve, and delete data according to keys.
The thesis will conclude with an evaluation of the objectives, benefits, and shortcomings of the proposed and implemented library. Possible directions for further development and improvement of the library will also be proposed.

Bibliography / sources:

Steen M., Tanenbaum A. S., “Distributed Systems”. 2023. ISBN 9081540637
Santoro, N “Design and Analysis of Distributed Algorithms”. Wiley, 2007, ISBN 0471719978
Tel, G. "Introduction to Distributed Algorithms". Cambridge University Press, 2001. ISBN 0521794838

Name and workplace of master’s thesis supervisor:

Ing. Peter Macejko   Department of Telecommunications Engineering FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **19.02.2024**   Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **15.02.2026**

Ing. Peter Macejko Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

Děkuji Českému vysokému učení technickému za celé dva roky které jsem na její Fakultě Elektrotechnické strávil. Děkuji, že se mi zde dostalo kvalitnímu vzdělání v oboru mého zájmu. Věřím, že znalosti zde získané, zkušenosti zde nabyté ale i strasti zde prožité mne učiní lepším vývojářem, kolegou a hlavně člověkem.

Jmenovitě bych chtěl poděkovat panu inženýrovi Peteru Macejkovi. Děkuji Vám za Váš čas, velkou snahu a trpělivost. Domnívám se, že Vaše snaha diskutovat nad tématem pravidelně se znatelně podepsala na kvalitě této práce.

Děkuji mé přítelkyni, bez které bych studium na ČVUT nezapočal.

Děkuji přátelům a mé rodině.

I thank the Czech Technical University for the two years I spent at its Faculty of Electrical Engineering. Thank you for providing me with quality education in my field of interest. I believe that the knowledge attained, the experience gained and the hardships lived here will make me a better developer, colleague and most importantly a better person.

In particular, I would like to thank Ing. Peter Macejko. Thank you for your time, great effort and patience. I believe that your efforts to discuss the topic on a regular basis have noticeably contributed to the quality of this work.

Thank you to my partner, without whom I would not have started my studies at CTU.

Thank you to my friends and my family.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 24. května 2024

I declare that I have prepared the submitted thesis independently and that I have cited all the literature used.

Prague, 24. May 2024

Abstract

This thesis discusses and describes an implementation of three basic types of distributed hash table (DHT). The purpose of this implementation is to use it to illustrate and teach distributed systems.

In the first part, the thesis describes the definition and mechanisms of a distributed hash table theoretically. Three basic DHT types are introduced - Chord, Pastry and Kademlia. The types are also compared. Next, two commercial solutions that use the technology are briefly introduced.

The second, mostly practical part, takes the reader through the process of implementing the selected DHT types. This part also highlights the pitfalls that can arise when implementing these systems, thus complementing the original publications which introduced these technologies originally. The three selected DHT types are further wrapped into a unifying library that allows the creation, customization, and management of each DHT type. The paper concludes with an evaluation of the results and suggestions for future work.

Keywords: Distributed Hash Table, Consistent hashing, Chord, Pastry, Kademlia, Java, gRPC

Supervisor: Ing. Peter Macejko

Abstrakt

Tato diplomová práce pojednává a popisuje implementaci tří základních typů distribuované hashovací tabulky (DHT). Smyslem této implementace je využít ji pro ilustraci a výuku distribuovaných systémů.

Práce v první části teoreticky popisuje definici a mechanismy distribuované hashovací tabulky. Představujeme zde tři základní typy - Chord, Pastry a Kademlia. Typy jsou taktéž porovnány. Dále jsou stručně představeny dvě komerční řešení které technologii využívají.

Druhá, převážně praktická část, čtenáře provede procesem implementace zvolených typů DHT. Tato část také upozorňuje na nástrahy které mohou vyvstat při implementaci těchto systémů, čímž doplňuje původní publikace představující tyto technologie. Tři zvolené typy DHT jsou dále obaleny do jednotné knihovny umožňující vytvoření, kustomizaci a správu jednotlivých typů DHT. Práce končí vyhodnocením výsledků a úvahou nad návrhy budoucích prací.

Klíčová slova: Distribuovaná hashovací tabulka, Konzistentní hashování, Chord, Pastry, Kademlia, Java, gRPC

Překlad názvu: Implementace DHT v Javě

Contents

Figures

Tables

May 24, 2024

Chapter 1
Thesis overview

In the era of distributed computing, the management and retrieval of data across a network of interconnected nodes pose significant challenges. Addressing these challenges requires robust and scalable solutions, and Distributed Hash Tables (DHTs) emerge as key components in the design of efficient distributed systems. This thesis delves into the intricacies of various DHT implementations.

Aim of the thesis is to not only explain the intricacies of these complex systems, but to also design a tangible playground for newcomers in form of a real implementation. The main output of this thesis is a general Distributed Hash Table interface, with the ability to create a new network or to connect to an already existing one. This interface can be then used to educate students and other developers. Furthermore, the interface can also be extended in future works to make a visual representation of such distributed system, therefore further facilitate the intuitive learning.

Thesis consists of three main parts. The first part introduces reader into the topic of distributed hash tables. We start with a general explanation of a distributed system and characteristics. We then introduce a *lookup problem* in context of such systems and present the distributed hash table as a solution to this problem. Chapters following the problem statement introduce three main types of DHT: Chord, Pastry and Kademlia. Analysis of each type follows in order to assess and compare their characteristics. The theoretical part then concludes with a research of current commercial DHT implementations.

Second part leads the reader through a real process of implementing three previously described DHTs. It provides further exploration of its concepts and practical utilizations. Each chapter is followed by a proposal for future works focused on visualizing the protocol mechanisms in real-time. This hands-on part also contributes to the scientific and developer community by further describing each implementation. It not only adresses pitfalls that can occur during an implementation but also sheds light on topics and details that were not explained in the original publications.

The final chapter of second part concludes the work and presents its results. Summarization and assesment of implementation is presented here as well as suggestions for future work.

Part I

Theory

Chapter 2
Distributed Hash Tables

A distributed system is a collection of independent computers that appear to its users as one computer [Tan07]. These systems have become popular not only because of their simple interface perceivable to the user, but also because of their high scalability. This, coupled with their ability to distribute workloads efficiently and provide fault tolerance makes distributed systems a very attractive choice for many technology companies no matter their size.

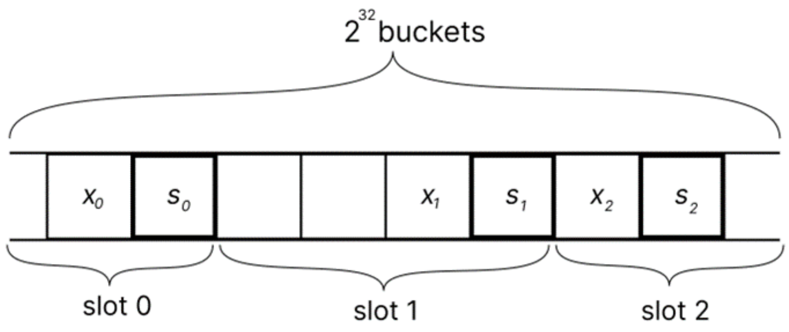
Among different architectures of distributed systems, two architectures are very prominent at the present time: client-server and peer-to-peer. Although both share similar properties of distributed systems, peer-to-peer architecture (p2p) aims to eliminate any requirement for separately managed servers and their associated infrastructure. This characteristic tends to be very advantageous considering the fact that the performance difference between personal computers and servers narrows day by day as well as network connectivity and speed increases [Cou11]. Other authors also point out increased resilience to single point of failure given by the highly distributed nature of p2p networks. On the other hand, the implementation of a p2p system may entail higher degree of complexity.

This thesis focuses on one of many complexities called the lookup problem [Ste05]. Assuming that all participants in the network have equal privilege and no participant has knowledge of the whole network, the lookup problem asks: Where to store and find a certain item? While Steinmetz provides three [1] possible solutions to the problem, the thesis at hand explores perhaps the most elegant one, the distributed hash table.

2.1 What is a Distributed Hash Table

The distributed hash table extends the basic functionality of a plain hash table by distributing key/value pairs across a network of cooperating machines (nodes). This results in decreased lookup time, as well as an increased amount of data that can be stored in a DHT. It comes from the fact, that the amount of data stored is limited by the total sum of space available on the nodes

[1]Steinmetz compares three approaches: centralized server, flooding search and distributed indexing.



DHT space partitioning assigns ownership of buckets in slots to one particular machine. Partitioning scheme in the figure corresponds to Chord's scheme where each value x_i is owned by the first following machine s_j to the right (machines are hashed to the DHT space as well). Source: author

Figure 2.1: DHT space partitioning

participating in the network. DHT also keeps the two basic hash table operations of put(key) to store and get(key) to retrieve a value [Gho06].

In order for a node to participate in a DHT, it only needs to keep track of its own data and manage a small number of routing references to other nodes in the network. Upon a lookup request on any node, the node itself either retrieves the item from the local hash table or routes the request to other nodes. The routing request is then propagated through the network until it arrives at the node that actually stores the item in its local hash table. DHT are usually very effective at routing. Depending on implementation, it usually takes O(log N) hops for a message to reach the destination, where N is the size of the network [Gho06]. Some DHTs are able to route in even less hops.

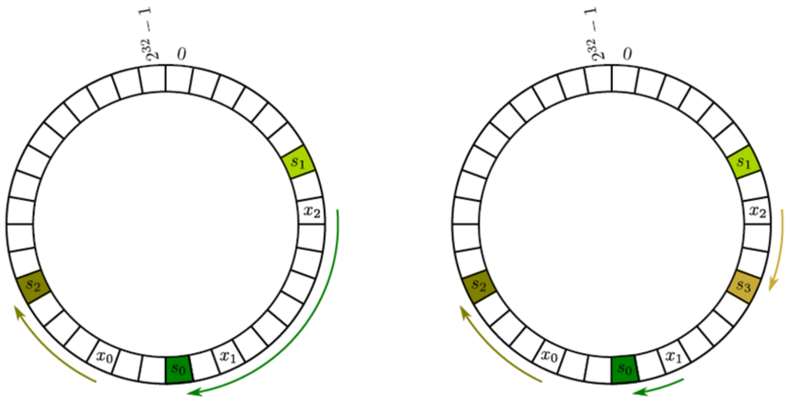
2.1.1 DHT space

DHT space is defined as a set of all possible IDs of items that a DHT can manage. This can be viewed for example as 32-bit value range (n-bit in general) or a set of 2^{32} buckets. Once an item is inserted into a DHT, it is mapped to a bucket. For many implementations, this mapping happens also when a new node joins the DHT as we will explain in the following chapters. Both mappings are result of a global mapping function that the DHT system uses [Zha13].

DHT space partitioning

In order for DHT to be scalable, it distributes stored values over many machines participating in the network. Keyspace partitioning is a scheme which globally assigns ownership of values to given machines. This in turn allows efficient storage and lookup in decentralized system.

Different DHT implementations do not follow a common general partitioning principle. Chord and Pastry use *consistent hashing* algorithm which stores



Join of machine s_3, value x_2 is reassigned to the newly joined machine. Source: [Rou22]

Figure 2.2: Responsibility transfer

items on circular id space with defined ownership slots [Zha13]. Kademlia stores items in binary tree structure which implicitly divides stored items in slots aswell. Then there is also rendezvous hashing, achieving same thing through different means. It is however helpful to at leat first introduce consistent hashing to get the intuition of keyspace partition since Chord and Pastry build upon it. We therefore dedicate the following chapter to consistent hashing while Kademlia partitioning will be studied later. Rendezvous hashing is beyond the scope of this thesis.

Consistent hashing

To map a value to a bucket, we could use a hash function in the form of h(x) mod n where x is the value and n is the number of machines in the network. This however wouldn't scale well if we were to change the number of machines n frequently. Let us suppose a machine failure and therefore number n changing to n-1. We would end up with faulty mapping of almost all values. Consistent hashing algorithm solves this problem by hashing machines to the same DHT space as values. By placing n machines into the keyspace we inherently split it into n slots, each slot owned by the node at the end of the slot. Or rather each value is owned by the first machine found when traversing the keyspace to the right. If we encouter the end of the space before reaching the owning machine, we just wrap around the space. The keyspace is therefore said to be circular [Rou22].

Upon insertion of item x, into a bucket $h(x)$, we traverse the keyspace until we find a bucket $h(s)$ onto which a node was mapped previously. We then exclaim that such node s is responsible for managing item x and store it locally on the node s [Rou22].

This way of item and node mapping yields two very appealing properties [Rou22]:

1. We can expect that there is no hotspot in the network and each server stores roughly 1/n objects[2].

2. The entire keyspace does not have to be rehashed upon a new node arrival. Set of items that need to be remapped is the first half of the arc upon which the node gets inserted. This is again only 1/n fraction of objects.

2.2 Chord

Chord was one of pioneering DHT implementations proposed in 2001. It's aim was to tackle a couple of p2p systems difficult problems, namely *load balancing*, *decentralization*, *scalability* and *availability*. To address these problems, it builds on the concept of circular consistent hashing [Zha13].

Chord's DHT space is a circle, refered to as *Chord ring*. IDs for nodes and values can be computed by applying a hash function. IDs are then arranged ascendingly in clockwise manner on the Chord ring. Nodes on the ring maintain specific ranges of data defined by the id of the responsible node and its predecessor. Ranges are kept in so called *finger tables* on each node. These allow for very efficient lookup in $O(\log n)$ time complexity where n is the number of nodes currently online in the DHT [Zha13].

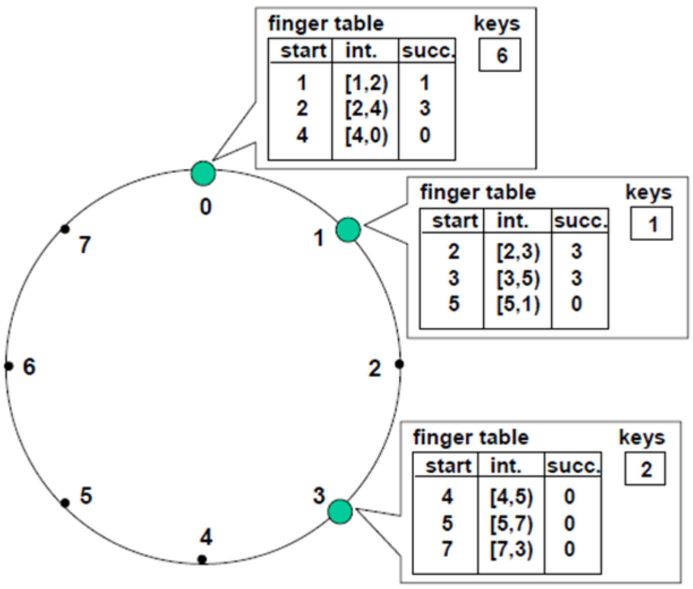
2.2.1 Finger table

To implement consistent hashing in distributed environment, one only needs to store very little volume of routing information. It should be enough for a node to only know about it's immediate successor. This approach on its own would not be very efficient however. We would need to make n hops in the worst case to get the correct mapping. To make the lookup faster, Chord implements additional routing table called *finger tables* [Sto01].

Each node keeps at most m entries in its finger table, where m is also the number of bits in key/node identifier. The ith entry (or *finger*) on any node A refers to a node B, which is at least $2^{\text{i-1}}$ further on the identifier circle, therefore $\text{id}_\text{B} >= (\text{id}_\text{A} + 2^{\text{i-1}}) \bmod 2^m$, where i is positive non-zero integer less or equal to m.

Let's briefly consider node with id value of 1 on the Figure 2.3 below. It's second entry in finger table refers to a node that is at least $2^{\text{2-1}}$ values further on the identifier circle. This way there is only little information stored on each node. Each node also knows more about nodes that are closer rather than further. Also note that any finger table on its own does not contain enough information to determine the successor of any possible key [Sto01].

[2]Assuming the chosen hash function provides random distribution.



Hypothetical instance of Chord. There are three nodes displayed along with their respective finger table. Source: [Sto01]

Figure 2.3: Finger table example

2.2.2 Find successor procedure

Finger table is also used in the case that a given node does not know the successor of a key that is being looked up. Consider key k is being looked up by node A and node A does not know k's successor[3]. In that case, A will look up from it's finger table a node X such that id_X precedes id_k more immediately then id_A. Node A will then require node X to again respond with id of node that precedes id_k more immediately then id_X. This way node A gradually learns of nodes closer to k. Authors of Chord paper call this algorithm *find_predecessor()*. Note that *find_successor()*, which would return most immediate **succeeding** node, could be implemented on top of this algorithm, simply by returning the first successor of the found node [Sto01].

2.2.3 Node arrival

Chord is able to preserve it's ability to locate any key even after a node join. It needs however to also maintain two invariants to preserve it; succesor of each node must be correctly preserved and for each key k there must be a node which is its successor, and the node is responsible for key k. Each node in Chord also keeps a predecessor pointer to its immediate predecessor, which makes it easier for a node to join [Sto01]. Let us assume that we have a new

[3]Chord assigns responsibility for a key k to a node N such that idk   idN. The responsible node is therefore key's successor.

node X joining the network and X knows of a node A in the chord network in advance.

Initialize finger table of newly joined node

The first couple of steps are very similar to a node joining doubly linked list. Node X will set its first finger as A's first finger. Successor and predecessors are then updated on the neighbouring nodes. Finally, X will initialize its fingers by traversing the identifier ring forward, potentially asking node A to find a successor of node that X is unable to find on its own [Sto01].

Update finger table of other nodes

Newly joined node X will then have to be also entered into other finger tables. This is done by traversing the identifier circle in counter-clockwise manner. It can be proved, that there has to be only $\mathrm{O}(\log n)$ nodes updated after a single node join [Sto01].

Key transfer

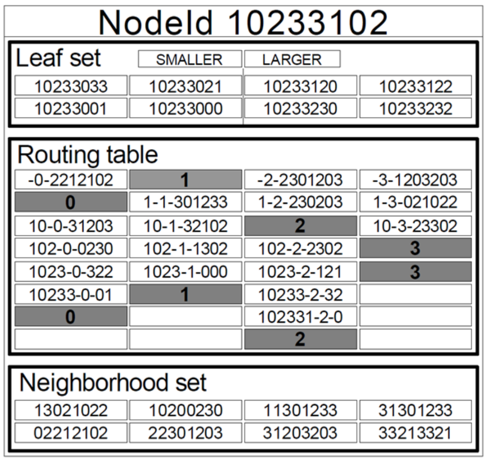
Responsibility for keys in the region of X's join must be reassigned as the final step. The nodes that need to be reassigned are those that X is now succeeding on the identifier circle. Responsibility is transferred from the current immediate successor of X to X. The joining node therefore only needs to contact one node [Sto01].

2.3 Pastry

Since the proposal of DHT technology (or consistent hashing to be exact) in 1997 [Kar97], there had been many DHT implementations which set out to improve upon the technology, be it in terms of better node locality, routing, security or performance [Sto01], [May02]. Pastry is a DHT implementation that aims to improve the lookup process in terms of a scalar proximity metric that the user specifies [Row01].

Authors of Pastry implementation set out to improve upon the ability of a peer-to-peer network to locate objects and provide efficient routing algorithms while preserving the common properties peer-to-peer systems [Row01].

Pastry uses a heuristic to guarantee delivery of message in less than $\lceil \log_{2^b} N \rceil$ routing hops, where N is the total number of nodes in the network and b is base of the identifier space. Each node is allowed to do message-specific computation. Furthermore, nodes are notified upon each node failure, join or recovery. Global random node id assignment ensures diversified geography, ownership, jurisdiction etc. Eventual delivery is guaranteed under the assumption, that no $\lfloor \frac{|L|}{2} \rfloor$ nodes with adjacent ids fail at the same time, where |L| is configuration parameter frequently set to 16 or 32 [Row01].



NodeId 10233102

Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Node state of a hypothetical node. Ids are in base 4 format. NodeIds in Routing table are in a format of `prefix - next digit - rest of id`. Shaded cell displays respective digit in the current node's id. Source: [Row01]

Figure 2.4: Pastry's node state

2.3.1 Design

For a machine to join Pastry network, it only needs to run the Pastry node software. Upon joining the network the machine is assigned 128-bit identifier refered to as *nodeId*. NodeId then also indicates the position of the machine in the DHT space [Row01].

The following paragraph summarizes routing which is mostly dependent on *pastry node state* described in the next chapter. When routing, nodeId is perceived just as digit sequence stored in *routing table* of each node. Numbers in the sequence are of base 2^b. This sequence then determines where to route incoming messages. Let's assume a message addressed to node D arrives onto node A. A looks up its routing table and compares digit sequence of id_D with sequences in the routing table. Node A then forwards the message to a host X such that id_X has longest prefix match out of all hosts with id_D. If there are more hosts with such a match, message is forwarded to numerically closer id to id_A [Row01].

2.3.2 Node state

State of Pastry node is defined as "at the time" configuration of three sets:

1. Routing table
2. Neighborhood set
3. Leaf set

The routing table on each node has exactly log N rows. Let's assume routing table R_A owned by node A and it's nodeId id_A. An entry on the nth row of R_A holds nodes that share first n digits in its id with id_A. On the first row, there would be ids sharing only the first digit, on the second row, there would be ids sharing the first two digits and so on (there is also 0th row sharing no common prefix). There are 2^{b-1} of such entries on each row. The parameter b is a configuration parameter that determines size of populated portion of routing table and maximum number of hops. Although there may be many potential entries with corresponding prefixes in each row, we only fill it with ids of nodes that are close to node A in terms of chosen proximity metric. If there is no such node, the entry may be left empty [Row01].

The neighborhood set M contains the set of closest neighbouring nodes according to a chosen metric. It does not usually participate in routing although it comes into play when we want to maintain locality properties [Row01]. Lastly, the leaf set L contains nodeIds that are numerically closest to the current node. One half of leaf set contains nodeIds larger than current nodeId and the other half contains node smaller than current nodeId. Leaf set directly participates in message routing [Row01].

2.3.3 Routing

The routing procedure is done every time that a message addressed to node D arrives to a node A. Listing 2.1 describes the whole procedure. Variables are decribe as such:

- *Lmin, Lmax* – ids of lowest and highest value in leaf set L
- *R[i][j]* – entry on the ith row and jth column
- *idD[i]* – ith digit in D's nodeId
- *shl(idD, idA)* – returns common prefix lenght of both id

We begin by looking up the recipient of message in the leaf set. If id_D is in range of ids contained in leaf set, we forward the message to a node with numerically closest id to id_D. If there is no such id in the leaf set, we start looking up the routing table. We try to lookup and forward to a nodeId which shares the prefix by at least one more digit then the current node. If there is no such id in the routing table, we forward to an id which has same prefix length and is numerically closer t to id_D than current id id_A. Such routing procedure converges since on each routing the message is either sent to a node with longer prefix or is at least numerically closer [Row01].

Listing 2.1: Pseudocode of Pastry routing procedure. Source: Row01

```
if(Lmin <= idD && idD <= Lmax) {
    // leaf set
    forward to L[i], such that:
        L[i] is numerically closest to idD
} else {
    // routing table
    l = shl(idD, idA)
    if(R[l+1][idD[l]+1] != null) {
        forward to R[l][idD[l]]
    } else {
        forward to T, such that:
            (contains(L, idT) || contains(R, idT) || contains(M,
                idT))
            && shl(idT, idD) >= l && |idT-idD| < |idA-idD|
    }
}
```

2.3.4 Node arrival

Initially, a new node needs to inform the network of its arrival as well as initialize its own state tables. It is assumed that the new node X knows of a nearby node A which already participates in the network. Some sources refer to node A as *bootstrap node*. Node X requests node A to send special „join“ message through the network. Message's destination is set to id of node X, therefore it arrives to numerically closest node Z. Nodes along the path of the „join“ message are requested to send their node state back to node X [Row01].

Neigborhood set of node X is initialized to A's neighborhood set since they are in proximity of each other. X's leaf set is initialized to Z's leaf set since its id is closest to X. Initialization of routing table is not so trivial however [Row01].

Let's describe the most general case in which A and X share no common prefix. In this situation there are no ids with a common prefix in row zero (A_0) of A's routing table. Therefore, entries in A_0 are also appropriate for X_0. A_1 however contains no entries that would be relevant for X_1. Nonetheless there are relevant entries in B_1, where B is a node that the „join“ message was routed to from A. This is implied by the fact that messages are routed to a node with at least one more common digit. To fill entries of row X_2 we again lookup the entries in C_2 and so on. Once the routing table of node X is filled, X informs of its state all the nodes in set $S_X \in (M_X \cup R_X \cup L_X)$. Nodes in this set then update their state accordingly. At this stage, X is finally able to fully participate in the network [Row01].

2.3.5 Locality

As stated above, Pastry DHT was designed to maintain locality. Authors define this property as following: *all routing table entries refer to a node that is near the present node, according to the proximity metric, among all live nodes with a prefix appropriate for the entry*[4] [Row01].

In order to demonstrate maintenance of locality, we will assume Euclidean proximity metric and that the locality property holds before node's X arrival to the network. As mentioned in the previous chapter, new node X knows of a nearby node A prior to joining. Trivial observation reveals, that entries in A_0 are close to A, A is close to X, and since triangulation holds, entries in A_0 are close to X as well. Locality is therefore preserved [Row01].

Now we turn our attention to X_1 and B (recall that B is second node en route of X's join request). Entries in B_1 are close to B, we don't know however how close B to X is. Therefore, it seems that setting X_1 to B_1 would not preserve locality. Contrary to this intuition, entries are quite often close. This comes from the fact that entries in each following row are chosen from exponentially decreasing set. B_1 is consequently an appropriate choice for X_1. The same can be said about each following level [Row01].

2.4 Kademlia

Compared to previous implementations, Kademlia can offer some interesting properties that other DHTs cannot. These are:

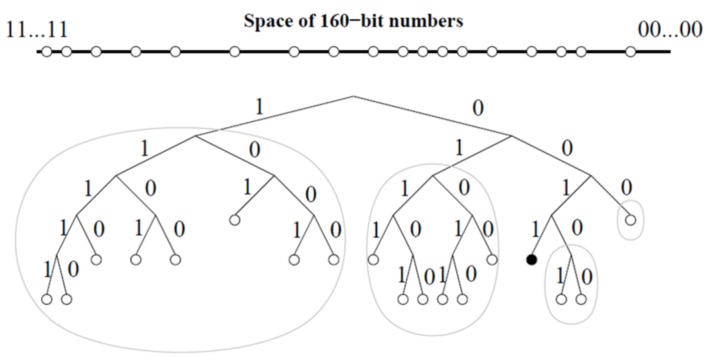
- Minimization of config messages
- Nodes are able to route through low-latency links
- Usage of asynchronous parallel queries

Moreover, Kademlia introduces a novel metric of distance based on exclusive OR operation. This metric allows for nodes to learn routing information from messages they receive. Routing itself is done in a similar way as with other implementations. It first cuts out a sizeable portion of DHT space that the node of interest does not occupy and converges onto the node in logarithmic time [May02].

2.4.1 System decription

By default, Kademlia provides 160-bit ID space. Both the nodes and values have assigned ID from this space. To better ilustrate a Kademlia instance, authors of Kademlia paper think of it as a binary tree, where each node keeps references to subtrees which do not contain the given node. A reference points to a single *contact* node. This in turn guarantees that any two nodes can find each other within a Kademlia instance.

[4]Every routing entry in the DHT must refer to such node. Statement "prefix appropriate for the entry" simply means that the prefix matching condition is satisfied, thus the entry not only satisfies prefix matching but the proximity as well.



Black dot shows a given node with ID of 0011. Subtrees which do not contain 0011 are circled. Node 0011 must have a reference to a contact node in each of them. Source: [May02]

Figure 2.5: Kademlia identifier space

When a given node A is trying to locate the target node B, it repeatedly queries the best node that A knows of to find contacts in successively lower subtrees.

2.4.2 XOR metric

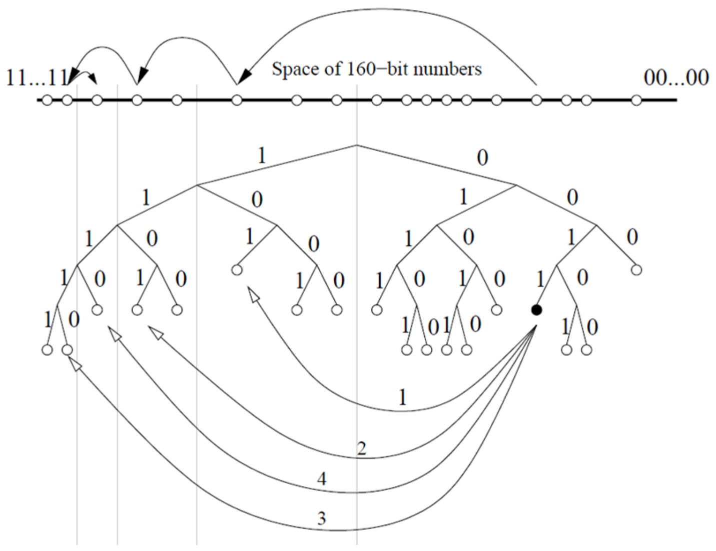
Unlike from other DHT implementations, sent messages contain ID of the sender. The recepient thus also receive the information about sender's existence. Values are assigned to a node based on distance metric which in Kademlia, is defined as exclusive bitwise OR (XOR) between two IDs. Couple of things can be observed from this metric:

1. $d(x,y) > 0$ if x and y are different nodes
2. $d(x,x) = 0$
3. $d(x,y) = d(y,x)$, therefore XOR is symmetrical
4. XOR follows the triangle inequality

XOR therefore offers features of distance metric while being cheap and simple. Note that XOR also captures the distance implied in binary tree.

2.4.3 k-buckets

To route messages, each node in the network must store contact references to other nodes. These references are stored in n number of k-buckets. The count of buckets on a node n is the number of bits in ID. System parameter k is chosen so that the probability of any k nodes failing in a short period of time is reasonably small. There can be at most k nodes in any k-bucket. The i-th k-bucket on any node stores references to nodes that have distance (in terms of XOR metric) in range from 2^i to 2^{i+1}. To better illustrate this, consider origin node 001101 (thick blue line in Figure 2.7) and target node 001100.



Given node 0011 is trying to locate node 1110. The best node it knows of is node 101, therefore 101 is contacted first. Following messages are addressed based on responses from contact nodes. ID space above shows converging onto the target node. Source: [May02]

Figure 2.6: Kademlia routing

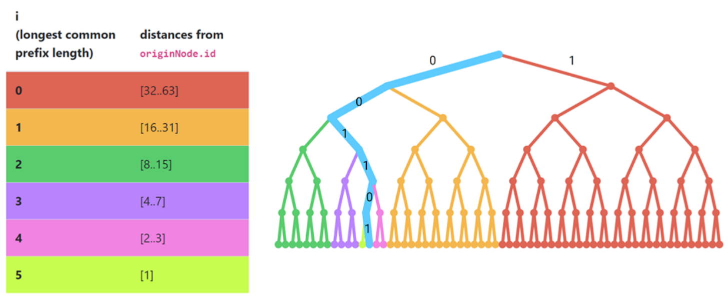
IDs of these nodes differ only in the last bit, therefore XOR distance is 1. If we consider target node 101101 differing in the first bit, the distance is 32.

2.4.4 Lookup

Kademlia protocol offers 4 distinct RPC calls:

1. PING – probes a node for liveness
2. STORE – prompts a node to store a given value
3. FIND_NODE – returns k nodes that are closest to a target
4. FIND_VALUE – similar to FIND_NODE but recipient returns value if it received a STORE call for the value previously

Lookup procedure is performed recursively, extracting succesively closer nodes to the given target on each iteration. Origin node first fetches number of nodes that are closest to the target from its point of view. It then sends parallel, asynchronous FIND_NODE calls to all the fetched nodes. Each recipient fetches its closest neighbors to the target node and sends them back to the origin node. Out of all nodes the origin has heard of, it again queries nodes that have not been queried before. This process repeats until there are no unqueried nodes coming back from recipients.



i (longest common prefix length)	distances from `originNode.id`
0	[32..63]
1	[16..31]
2	[8..15]
3	[4..7]
4	[2..3]
5	[1]

Visualization of distance of nodes in k-buckets. Source: `https://kelseyc18.github.io/kademlia_vis/basics/3/`

Figure 2.7: KBuckets

2.4.5 Join

Just like in implementations presented earlier, a joining node X must know of a node A already present in the network it wants to join. Upon joining, X inserts A to its own respective k-bucket, followed by FIND_NODE lookup with X's ID as the target ID. Node A will in turn refresh it's k-buckets that are further than closest neighbor. Finally, node X fills its own k-buckets as well as inserting itself into appropriate k-buckets on other nodes.

2.5 Comparison of different implementations

This chapter will summarize the research of various implementations of DHTs done so far. Analysis delves into diverse aspects, beginning with an examination of node design, wherein we closely examine the structural characteristics of individual nodes within the DHT Subsequently, we navigate through the intricate landscapes of topology, distance metrics, parametrization, routing procedures, and node arrival mechanisms. By comparing these elements across different implementations, this chapter illuminates the diverse strategies employed to achieve efficient and scalable distributed hash table systems, thereby offering valuable insights for the optimization and selection of suitable implementations in varying contexts.

2.5.1 Node design

After researching each node design, we conclude that the Pastry node may be the most complex one. Pastry node design harbors three distinct node reference sets (recall leaf set, neighborhood set and routing table), which can make it cumbersome for implementation. This design choice, however, seems to be providing a balance between local and global routing – while routing table provides links that may be further, neighborhood set contains the closest links. One should also recall that by "closer" and "further" means in terms of user specified metric. This scheme, while providing a useful feature, may also introduce some extra work since each node has to compute the metric for the linked nodes even if the metric is simple ping response time.

Chord node design on the other hand stores only one set of links, the finger table. This makes it lightweight. Apart from this set, Chord node may store two extra links for immediate predecessor and successor. There is no notion of scalar metric as in Pastry as well as no node specific computation.

Lastly, Kademlia nodes also appear to be also lightweight compared to Pastry. Out of the studied DHTs, Kademlia nodes are the only ones leveraging parallel asynchronous calls, making them a good match for today's machines. The use of parallel asynchronous calls enhances Kademlia's responsiveness, allowing nodes to efficiently handle multiple concurrent operations. However, it's worth noting that it also introduces a level of complexity that can make the implementation more challenging.

2.5.2 Topology

Chord network topology can be characterized as a simple circular keyspace. It maps values onto the keyspace in ascending clockwise order. Values are then stored based on the range in which they happen to be inserted. This range is owned by the first succeeding node. This node again marks the start of the next range owned by first succeeding node of values in this range. Kademlia and Pastry, however, both form hierarchical tree-like structure. While Kademlia places nodes onto a literal binary tree, Pastry organizes nodes based on common prefixes. Some sources refer to this topology as circle with tree topology. Recall that Pastry node initially looks up nodes in leaf set, where nodes with numerically closest ids are stored, if no suitable nodes are found (hierarchical structure), we look up the routing table (circle) [Elb15, Zha13].

2.5.3 Distance

All DHT implementations revolve around distance metric. The metric expresses the distance between two objects in a distributed system and is therefore a central part of both the routing and retrieval operation. Different metrics thus imply different routing strategies. Kademlia takes advantage of bitwise exclusive OR operation when computing distance. Not only is this operation very cheap, but it also does not require any other algorithmic structure. Pastry measures distance between two objects by matching each digit in their ID. Finally, Chord defines distance from A to B as $(id_B - id_A) \bmod 2^m$ where m stands for number of number of bits in the id [Zha13].

2.5.4 Parametrization

Since Chord uses hash function to generate node identifiers, we can pick one suiting our needs – either one that generates IDs faster or distributes them evenly.

Kademlia offers two parameters which directly influence the efficiency of routing. The size of k-buckets k should be chosen to reasonably high number so that there is small probability of any k nodes failing within an hour. Second parameter is , a system-wide concurrency parameter. Anytime Kademlia node is looking up a certain object, it contacts nodes at once.

Pastry DHT can be customized in terms of minimum simultaneous adjacent node failures denoted as |L|. Pastry guarantees eventual delivery if there is no more than $\lfloor \frac{|L|}{2} \rfloor$ simultaneous node failures. One can also configure the base parameter **b**. Base determines the range of values of single ID digit. Routing table also contains **b** entries in each row. It therefore directly influence the connectivity of the network.

2.5.5 Routing procedure

This chapter will describe the routing procedure in each implementation. We assume a node A looking up destination id d stored on node D.

Recall that in Chord, node D is the most immediate node successor after d on the chord ring. In case that node A does not know d's successor, A will lookup node X, such that id_X precedes d more immediately then id_A. Node A then requires X to lookup Y, such that id_Y precedes d more immediately then id_X. Finnally we converge to D's predecessor, it's successor stores d and we've just found it.

Pastry stores d on node D such that id_D matches d's prefix the most out of all live nodes. Node A will therefore initially lookup its leaf set if d is in it's range. It forwards the message to the numerically closest nodeID if d is in the range. Otherwise A will lookup routing table for id sharing at least one more digit with d than id_A does. If this fails as well, A will lookup id which shares same amount of digits but is numerically closer to d. Therefore procedure converges on longer prefix or numerically closer node.

In Kademlia node A will fetch number of nodes that are closest to d. A does the lookup from its closest non-empty k-bucket. It then proceeds to send parallel asynchronous calls to all fetched nodes. Its recipients then also fetch the same number of nodes in the same manner, informing A of nodes closest to d from their point of view. A will then query those nodes and repeat the process until there are no unqueried nodes coming back from recipients.

2.5.6 Node join

All implementations require a bootstrap node A (node already participating in the network) to help newly created node X join the network.

As the first step, Chord node X will copy A's first finger and update predecessor and successor. The rest of X's fingers are initialized by traversing the chord ring forward. Next step is inserting X into other finger tables which is done by traversing chord ring backwards. The last step is to reassign items in the range that had just been split by the join. Particularly those that are now newly succeeded by X. Node X's successor transfers this responsibility to X.

In Pastry, newly joined X requests A to send join message through the network. The destination of this message is set as X thus it arrives to a node Z which is numerically closest to X. Nodes along the path the send back their state to X. Node X will then copy A's neighborhood set and Z's leaf set. X's routing table will fill entry by entry from each node along the path.

After joining Kademlia network, X will appropriately insert A into X's k-bucket. X will then initiate FIND_NODE procedure on its own ID, this will refresh A's buckets. Finally, X will fill its own bucket as well as insert itself into other buckets.

2.6 DHT implementations in practice

Although there are almost none commercial DHT libraries currently on the market per se, there are many products based on DHT technology that target broader usage in distributed systems market. Following chapter offers an overview of two such technologies: Riak and Amazon's Dynamo (not to get confused with sister project DynamoDB). Overview of Dynamo is considered to be comprehensive while Riak's overview does not go in same depth. Surprisingly, Amazon engineers contributed to the academic domain by publishing their findings, whereas Riak developers did not. Riak authors therefore provide just a brief technology overview on their website.

2.6.1 Riak KV

Marketed as *NoSQL database*[5], Riak KV aims to provide a solution for *Big Data applications*[6]. It was created with high availability as a priority and provides simple key-value pair storage.

Architecture

Riak groups nodes into clusters. Nodes within a single cluster communicate with each other in order to provide data availability and partitioning. Each node within cluster is able to serve read and write request.

Clusters of nodes are placed on a common circular hash space called Riak Ring and just like in Chord, the last value is adjacent to the first. The ring state is known to all the nodes on the ring. Each time a nodes gets a request for an object that it does not manage, it forwards the request to the appropriate node. Upon any node join or failure, the participating nodes adjust and balance the partitions around the cluster, updating the whole ring

[5]NoSQL databases, or "Not Only SQL," are a class of database systems offering high scalability and flexibility. Unlike traditional relational databases, NoSQL databases are schema-less, accommodating unstructured or semi-structured data.

[6]A "big data application" refers to software designed for processing and deriving insights from vast and diverse datasets. These applications employ distributed computing frameworks to handle large volumes of data

state in turn. Furthermore, each node within a cluster may manage one or more virtual nodes.

Riak stores key-value pairs in *buckets*, where key is a binary value uniquely identifiyng a value whereas value is data associated with key. Riak stores objects in virtual namespace called buckets. Riak object can therfore be defined as three-pair combination of those. User can define custom bucket types with certain properties. This enables the user to manage a group of buckets since a bucket type inherits its properties [Bas16].

Replication

Consistent hashing distributes data across all nodes uniformly. Upon any join or failure, consistent hashing remaps only $\frac{K}{n}$ keys, where K is number of keys and n is the number of slots. Scaling can be done with almost no downtime since data is rebalanced in a non-blocking operation.

Data replication guarantees the presence of multiple data copies distributed across various servers. By default, Riak distributes three copies, each on a unique server. This results in high availability with low latency, since if any of the three replication nodes would fail, the request would still be served by other two. By simultaneous replication and partitioning, Riak KV creates horizontally scalable system [Bas16].

Eventrual consistency

Eventual consistency is one of distinct features of NoSQL systems. Eventual consistency means that a transaction in a distributed system is considered complete even if not all assigned nodes have confirmed it. Confirmation from all assigned nodes is therefore not required for the system to acknowledge the transaction's completion. This enables greater degree of concurrency as well as higher data availability. Riak KV could therefore be placed in AP intersection in *CAP theorem*[7] diagram. Replicas are eventually consistent, meaning they are always accessible, though not all of them will have the most recent updates. This creates inconsistent system states [Bas16].

Version conflicts, consistency recovery

Inconsistecy may arise upon a concurrent access to a single object or under a heavy load. Consistency is achieved whenever Riak receives a lookup request. It will search for all the object replicas and solve the issue by returning the most recent version by reading *Dotted Version Vector*[8] stored on each

[7]CAP theorem asserts that in a distributed system, it is impossible to simultaneously guarantee Consistency, Availability, and Partition Tolerance.

[8]Dotted Version Vectors are a concurrency control mechanism in distributed systems, tracking causality and ordering of events. They efficiently manage conflicts by incorporating a compact representation of causally-related updates, facilitating accurate reconciliation in distributed databases.

replica. Furthermore, Riak employs two strategies to resolve diverging replicas [Bas16]:

1. Read & Repair – synchronize all replicas during read operation
2. Active Anti Entropy – detecting divergency by repeatedly comparing Merkle trees over replicas

2.6.2 Amazon's Dynamo

A predecessor project of today's popular DynamoDB from the same company, Dynamo was created to be always available since for Amazon;

> Even the slightest outage has a significant financial consequences [DeC07].

Amazon's platform provides services to other web sites worldwide. This requires an infrastructure of many servers distributed all around the world. With system running on such scale it is almost inevitable for any component not to fail. At the same time, the platform must provide „always on" experience. For example, a customer should be able to add new items to his shopping cart even if a data store was just destroyed by a tornado. The system therefore required to treat failure as a normal case. Dynamo was developed to address these requirements. It uses a mixture of well-known techniques for achieving high scalability and availability [DeC07]:

- Data are partitioned and replicated using consistent hashing
- Consistency is achieved by quorum-like technique and a protocol for synchronization of decentralized replicas
- Nodes join and leave with no manual administration

Previous work and distinctive features

While there are multiple solutions in peer-to-peer software domain dedicated to address the problems of distributed data storage, none seemed to fit Amazon's needs. Even though Pastry and Chord offers comparable functionality to Dynamo, their multihop routing procedure was not enough for Amazon's needs. Freenet and Gnutella were other similar distributed storage systems. Search queries of these systems are however implemented by flooding the network to find nodes on which the data are stored.

Dynamo on the other hand tries to provide „always-writeable" data store, a system in which no write is denied. The motivation comes from the fact, that this could lead to poor customer experience. For example, the platform must allow the customer to add or delete items from his shopping cart even during network or node failures. Dynamo was also built under the assumption that it will be deployed on an infrastructure inside a single administrative domain where every node is trustworthy. Another requirement was to complete 99.9% of read/write operations under couple of millisecond. Dynamo therefore avoids multihop routing typical for Chord or Pastry. Authors describe Dynamo as a zero-hop DHT where every node tries to store just enough routing information to be able to route a request to the destination directly [DeC07].

Interface

As many other DHT's Dynamo offer simple get-put interface:

- *get(K)* – fetches object replicas associated with key K and return object or list with confliction versions and context
- *put(K, context, object)* – determines the placements of replicas based on key K and writes replicas to a disk, context is stored with the object and is later used to check the validity Dynamo applies MD5 hash on key K in order to generate 128-bit identifier. Identifier is then used to determine which nodes will manage the key [DeC07].

Partitioning

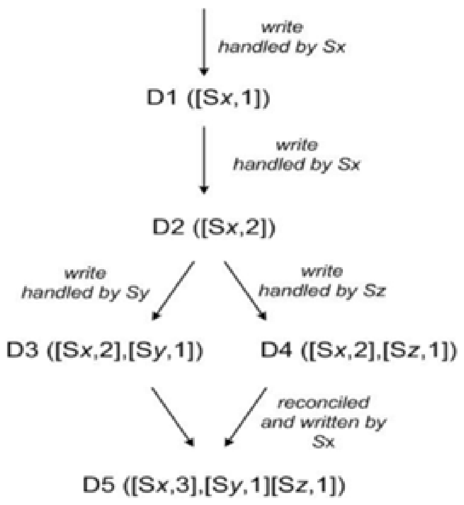
Dynamo replicates data across multiple nodes to achieve high durability and availability. Each data item is replicated to N nodes, where N is a configuration parameter. Replication is managed by a *coordinator nodes* (node that is handling the read/write operation). Each node is then responsible for range of value between itself and Nth predecessor.

Data versioning

Dynamo provides eventual consistency, allowing for concurrent updates to be propagated asynchronously. The *put()* operation, may return to the client even before the update was written to all the replicas. This may create a situation in which subsequent *get()* returns object without the most recent changes. The time to propagate changes is finite under no node failure assumption. When there are failing nodes, updates may not arrive to all replicas however.

Such scenario can be illustrated on shopping cart application. The cart application requires that the „add to cart“ operation was never forgotten or denied. Node failures may introduce divergent versions of same objects across its replicas. Updates in a network with partitioning and node failures can therefore create a scenario in which an object has two versions of „sub-history“ branches which system has to reconciliate. In order to reconciliate the branches, Dynamo uses *vector clocks* to capture causal relation between versions of a given object. A vector clock is a simple node-counter pair. It allows Dynamo to determine if two sub-versions are on parallel branches or have causal relation between themselves. If the counter on the first object is less than or equal to objects on other node, it means that the first object is an ancestor of other objects and therefore can be forgotten.Otherwise object changes are in conflict and therefore need reconciliation.

Let's assume a client initiates the creation of a new object, *D1*. This request will be handled by node *Sx*. Node *Sx* will therefore write the new



Reconciliation in Dynamo, Source: [DeC07]

Figure 2.8: Reconciliation in Dynamo

object across appropriate replicas and increase the counter. The system now contains object *D1*, along with its vector clock ([*Sx*, 1]). The client subsequently updates *D1* again and the request is handled by *Sx* as before. *Sx* therefore updates replicas and increments the counter, resulting in object *D2* with a vector clock of ([*Sx*, 2]). There are now two objects in the system, where *D1* is the direct ancestor of *D2*, which can be determined from their vectors. However, there can be replicas in the system that have not yet seen *D2*.

Assume there is another client accessing *D2* and updating it. This request will now be served by a different server, *Sy*, which creates *D3* with a vector clock in the form of [(*Sx*, 2), (*Sy*, 1)]. Next, assume a different client updating *D2* as well but via a different server, *Sz*. Server *Sz* will create *D4* with a clock [(*Sx*, 2), (*Sz*, 1)]. Currently, there are objects *D3* and *D4*, and in them are changes that are not reflected in each other. Both versions, therefore, must be kept and presented to a client for reconciliation.

Finally, assume a client reading both *D3* and *D4*. The read will return merged context [(*Sx*, 2), (*Sy*, 1), (*Sz*, 1)]. If an update is done via server *Sx*, the new data will have a clock [(*Sx*, 3), (*Sy*, 1), (*Sz*, 1)]. Vector clocks, therefore, may grow a lot if many servers access the same object. This is not likely in practice, however, since writes are usually handled by a small number of preferred nodes [DeC07].

Get & Put execution

Get and put operations can be executed by any node in the network. Client can however choose two strategies to select the node:

- 1. Routing the request through a generic load balancer that will select the node based on load information.

- 2. Using partition-aware client library, routing request directly to the given coordinator. This results in lower latency by skipping potential forward hop.

To uphold the consistency between replicas, Dynamo uses quorum-like protocol. The protocol offers two configurable parametrs R and W, where R is the minimum number of nodes which must participate in a successful read and W is the minimum number of nodes which must participate in a successful write. If $R + W > N$, then latency is dependent on the slowest replica. N is therefore set to a smaller value to achieve lower latency. Any time a coordinator receives put request, it generate the vector clock for new version, writes it locally and then sends it to N number of close nodes. Write is succesful once W-1 nodes respond. Get works similarly: upon receival of get, coordinator requests all version of data for the given key from N close nodes. It then waits for N number of responses and then returns to the client. If coordinator receives multiple versions of data, it returns all versions without causal relation and divergent versions are reconciled [DeC07].

Part II

Analysis and Implementation

Chapter 3

gRPC

gRPC is cross-platform open source remote procedure call framework created by Google. Initially created as closed source framework Stubby, Google released a new open source in the following years. The result is now called gRPC and is widely used not just by Google, usually as communication facilitator between microservices. [Con24]

gRPC uses *Protocol Buffers*[1] as *Interface Definition Language*[2] and as a format for message exchange. gRPC allows client application to directly invoke a method on a server on completely different machine as if it was a local object. It is based around a definition of a service and its methods (including parameters and return types). Server then implements this interface and runs the gRPC server to handle client calls. Client-side offers a *stub* object which provides the same methods as the server. [gA24]

3.1 How it works

The user first defines structures for the data that will be serialized in a *proto file* with a `.proto` file extension. Protocol buffer data are structured as messages. Each message is a small logical record of information containing *name-value* pairs called fields.

Listing 3.1: message.proto

```
message Person {
  string name = 1;
  int32 id = 2;
  bool has_ponycopter = 3;
}
```

The second step is to use *protoc* compiler to generate data access classes in specified language. gRPC services are defined in proto files as well with parameters of RPC methods specified as *protocol buffer messages*[gA24].

[1]An open source serialization method offering efficient encoding and language independence for structured data, widely used in distributed systems and communication protocols.

[2]Language or format for definition of interface used for communication between different software components independent of platform.

Listing 3.2: greeter.proto

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

3.2 Core concepts

Service definition

gRPC offers four types of service methods[gA24].:

1. **Unary RPC** - the client sends a single request to the server and gets back a single response

2. **Server streaming RPC** - the client sends a single request and gets back a *stream* from which it reads until there are no more messages. Preserved message order is guaranteed.

3. **Clieant streaming RPC** - the client writes a sequence of messages and sends them to server using provided stream, once the client is done, it will wait for server to read them and return a single response. Preserved message order is guaranteed.

4. **Bidirectional streaming RPC** - both sides are sending a sequence of messages using read or write stream. Streams operate independently and client and server can therefore read or write in any order they like. Message order is again guaranteed.

Listing 3.3: RPC_modes.proto

```
service ChordService {
    // Unary RPC
    rpc FindSuccessor(FindSuccessorRequest) returns
        (FindSuccessorResponse){}

    // Server streaming RPC
    rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse){}

    // Client streaming RPC
    rpc LotsOfGreetings(stream HelloRequest) returns
        (HelloResponse){}

    // Bidirectional streaming RPC
    rpc BidiHello(stream HelloRequest) returns (stream
        HelloResponse){}
}
```

Using the API

gRPC provides compiler plugins that generate client and server-side code. Users typically call these APIs on the client side and implement the corresponding API on server-side.

The server-side implements methods declared in server and runds the gRPC server to handle clients calls. gRPC infrastructure then decodes incoming requests, executes service methods and encodes the responses.

Client has a local *stub* object (some languages refer to it just as a client), which implements the same exact methods as the declared service. Client then calls these method on stub and wraps method parameters in corresponding protocol buffer message type. Request is then sent to a server and returns server's protocol buffer responses.

3.3 gRPC in action

3.3.1 Importing gRPC via Maven

As stated above, the first step is to declare the service and the request and response of its method using protocol buffers. The code snippet in listing 3.3 can be used for example.

To generate code in Java environment, we can use Maven's `protobuf-maven-plugin` which will run the protoc compiler with each `clean install`. For protobuf-based code generation integrated with the Maven build system, we can use `protobuf-maven-plugin` which will run the protoc compiler with each `clean install`. To include this plugin in our Java project managed with Maven build tool, we simply add this plugin into build section of `pom.xml`, see listing 3.4.

In order to include gRPC as JAR to a Maven project, we simply list the required JARs, these are in our case: `protobuf-java`, `grpc-netty-shaded`, `grpc-protobuf`, `grpc-stub`. Again, see the rough structure of `pom.xml` file in listing 3.4

Listing 3.4: pom.xml

```
<project>
...
    <dependencies>
   <!-- Protocol Buffers -->
       <dependency>
           <groupId>com.google.protobuf</groupId>
           <artifactId>protobuf-java</artifactId>
       </dependency>
   <!--transport-->
       <dependency>
           <groupId>io.grpc</groupId>
           <artifactId>grpc-netty-shaded</artifactId>
       </dependency>
   <!--a client-server common language-->
       <dependency>
           <groupId>io.grpc</groupId>
           <artifactId>grpc-protobuf</artifactId>
       </dependency>
   <!--client implementation-->
       <dependency>
           <groupId>io.grpc</groupId>
           <artifactId>grpc-stub</artifactId>
       </dependency>
   </dependencies>
...
   <build>
       <plugins>
           <plugin>
       <!-- Plugin to compile proto file into java files -->
               <artifactId>protoc-jar-maven-plugin</artifactId>
                   ...
               <configuration>
       <!-- The directory to search for proto files -->
                   <inputDirectories>
                       <include>src/main/resources</include>
                   </inputDirectories>
                   <outputTarget>
                   ...
       <!-- The directory to output the generated java files -->
                       <outputDirectory>src/main/java</outputDirectory>
                   </outputTarget>
               </configuration>
           </plugin>
       </plugins>
   </build>
</project>
```

3.3.2 Java interface

Creating a server

In order to create a gRPC server (assuming we have generated the Java base class), we first need to override the generated class alongside the methods we want to implement. In Java, This is usually done by simply creating a new class that extends the base class and overrides the methods. See listing 3.5 for reference. The method `findSuccessor` constructs a `FindSuccessorResponse`, returns it to client upon `onNext()` invocation and finishes the RPC with `onCompleted()` function call.

Listing 3.5: ChordServer.java

```java
public class ChordServer extends
     ChordServiceGrpc.ChordServiceImplBase {
    @Override
    public void findSuccessor(
        Chord.FindSuccessorRequest request,
        StreamObserver<Chord.FindSuccessorResponse> responseObserver
    ) {
        NodeReference n =
            ChordNode.this.findSuccessor(request.getTargetId());
        Chord.FindSuccessorResponse r =
            Chord.FindSuccessorResponse.newBuilder()
                .setSuccessorIp(n.ip)
                .setSuccessorPort(n.port)
                .build();
        responseObserver.onNext(r);
        responseObserver.onCompleted();
    }
}
```

Creating a client

To implement the client side which will invoke and consume the server's response, we first instantiate a gRPC channel with address on which the server is running in the format of `ip_address:port`. Using the channel, we then create the *stub* object. The stub object may be blocking or nonblocking. Since we are dealing with a simple RPC, a blocking stub will suffice. The request for the server is created next and it is sent to the server using the stub, by passing it as an argument to the method called on the stub.

Listing 3.6: ChordClient.java

```
public class ChordClient {

   public NodeReference findSuccessor_RPC(NodeReference n_, int
       targetId) {
      ManagedChannel channel =
          ManagedChannelBuilder.forTarget(n_.ip+":"+n_.port).usePlaintext().build();
      ChordServiceGrpc.ChordServiceBlockingStub blockingStub =
          ChordServiceGrpc.newBlockingStub(channel);

      Chord.FindSuccessorRequest request =
          Chord.FindSuccessorRequest.newBuilder()
             .setSenderIp(this.node.ip)
             .setSenderPort(this.node.port)
             .setTargetId(targetId)
             .build();

      Chord.FindSuccessorResponse response =
          blockingStub.findSuccessor(request);

      return new NodeReference(response.getSuccessorIp(),
          response.getSuccessorPort());
   }

}
```

Running a server

To actually run the server is pretty straight forward. First we need to build it using the provided builder with port on which we want our server to listen on and provide the instantiated service which we implemented in previous section. Next, we `build()` and `start()` the server. See listing 3.7 which displays a possible implementation which could be used to run Chord node as a server.

Listing 3.7: ChordNode.java

```java
public class ChordNode {

    private final Server server;

    public ChordNode(int port) {
        server = ServerBuilder.forPort(port)
                .addService(new ChordNode.ChordServiceImpl())
                .build();
    }

    public void start() throws Exception {
        server.start();
    }

}
```

Chapter 4
Chord

4.1 Chord node structure

Implemented node structure mirrors the one introduced by [Sto01]. It was decided to make one big `ChordNode.java` class, representing all logic that an actual Chord node serves rather then splitting the node to its *server* and *client* parts. Although this makes the node class lengthier, it matches the description of a node since it acts like server and client at the same time. It should be noted that this violates the GRPC recommended structure.

4.1.1 Predecessor, Successor and FingerTable

A simple `NodeReference` class was created with purpose to hold a reference to a single `ChordNode` instance. Within `ChordNode` itself, there is `NodeReference` representing current's node predecessor. There is also a list of `NodeReferences` representing the FingerTable.

Each `NodeReference` instance is defined by its *ip*, *port* and *id*, where *id* is output of SHA-1 hash of concatenated *ip* and *port*. Fields ip and port serve as GRPC handle for constructing GRPC channels.

4.1.2 Stabilization routine

Each ChordNode has an instance of `java.util.Timer` on which a stabilization `TimerTask` is scheduled. This timer is scheduled to run the stabilization task after a `STABILIZATION_INTERVAL`, which is a static variable with default value of 2 seconds. Stabilization routine is scheduled to guarantee the correctness of lookups. This is achived when nodes' successor pointers are kept up to date [Sto01].

4.1.3 GRPC server/client

As mentioned above, ChordService defined in `chord.proto` file was not splitted as per GRPC official recommendation. Both client and server methods were put into the same class instead to follow the natural logic of single Chord network node which requests as well as responds to other nodes. ChordNode class therefore contains GRPC Server field which implements all server-side logic within the same class as client-side logic.

4.1.4 LocalData in Treemap

The data managed by a Chord node instance is stored in a `java.util.TreeMap`. This choice of implementation, based on a red-black tree internally, was primarily made for its excellent support of range queries. Specifically, TreeMap provides the *subMap(fromKey, toKey)* method, which proves to be highly effective when a node needs to transfer responsibility for a range of its keys to a newly joined node.

4.2 Lifecycle

4.2.1 ChordNode instantiation

A ChordNode instance is created by simply calling the constructor and specifying node ip and port. Since by definition a single Chord node is it's own predecessor as well as successor, both are set to the node's own reference before any connection with another node is made. FingerTable is also filled with self reference.

4.2.2 createRing()

As per Chord's protocol specification, there should be a ring created first when starting a new Chord DHT instance. The *createRing()* method should be therefore called on a bootstrap node, onto which other nodes join.

The method takes no arguments and should be called on an instantiated ChordNode. After invoking the method, GRPC server is started on ChordNode's ip and port. ChordNode then starts stabilization thread invoked periodically. Stabilization involves fixing of random finger, checking for newly joined successor or potentially starting failure recovery sequence.

4.2.3 join()

After being instantiated, a node can join a running Chord ring by contacting a node already present in the ring. Upon joining, the node starts its GRPC server and initializes its neighbors and FingerTable with the assistance of the contacted node. It then requests existing nodes to insert the current node into their FingerTable. Subsequently, the node prompts its successor to transfer responsibility for the respective keys to the current node. Finally, the node initiates its stabilization thread.

4.2.4 leave()

The leave operation is analogous to join operation. First we stop the stabilization thread. Departing node will then prompt its immediate neighbors to update their respective successor and predecessor. Given departing node X, X will set its

- X.$s.p$ to X.p and
- X.$p.s$ to X.s,

where s is successor and p is predecessor. Finally, node X will transfer responsibility for its keys to its successor.

4.2.5 Node failure

Check for node failure occurs as part of every scheduled stabilization procedure. The GRPC call for successor's predecessor is sanitized for *unavailable* server state, which occurs upon GRPC server failure (successor's server failure). When this exception occurs, current node tries to find new online successor by traversing around the Chord ring, predecessor by predecessor. This chain call, initiated on node I, stops when predecessor of current node encounters predecessor which is offline as well. Let's refer to such predecessor as P. Since predecessor of P is the node that failed in the first place, P sets its new predecessor to node I and sends back its own address for node I to set P as initial node's new successor. Note that each predecessor along the way is prompted to modify its finger table entries to account for the failed node, thus replacing the failed node with its own address.

4.3 Logging levels

During implementation of Chord, the `SLF4J` was chosen as the main logging library. It was decided to split various events into three logging levels, depending on the abstraction layer of DHT system. First, there is `INFO` level, which when specified as the root level, logs information about the main DHT methods which are *put()*, *get()* and *delete()*. Then there is `DEBUG` level, logging useful information about Chord-specific procedures like *createRing()*, *join()*, *moveKeys()*, *leave()* and similar. Lastly there is `TRACE` level which is the most granular level, logging everything else which can be mainly used for developer's debugging purposes. Note when lower level is specified as root, it logs all levels above this one too.

4.4 Future work

When it comes to future endeavors, there lies a promising avenue for the visualization of steps encompassed within the debug-level procedures. Visualization can significantly aid in comprehending the intricate workings of Chord-specific operations such as *createRing()*, *join()*, *moveKeys()*, and *leave()*.

By creating visual representations of these procedures, developers can gain deeper insights into the inner workings of the Chord protocol. Visualizations can shed light onto relationships and interactions between Chord nodes an therefore facilitate a clearer understanding of system behavior.

Moreover, visualizations can serve as invaluable educational tools, helping newcomers grasp the fundamental concepts of distributed hash tables and Chord-based systems more intuitively.

Chapter 5
Pastry

5.1 Pastry node structure

The intial intention was to make Pastry implementation fairly similar to Chord. Later it was however decided to diverge from this idea since Pastry node structure is a bit more complex than Chord, especially considering the routing structures. All of these had to be locked to prevent concurrent access. To make the access to routing structures easier to work with, aside from big `PastryNode.java` class, another one was created, the `NodeState.java`, acting as a simple wrapper for the routing structures. This logical functionality split is congruent with previous Pastry protocol explanation where NodeState represented the routing structures as well.

5.1.1 NodeState

Each Pastry node has it's own NodeState with four main arrays representing the three sets of node references: upLeafs, downLeafs, neighborSet and finally routingTable which is a 2-dimensional array. Each array had to be locked to prevent concurrent access which could be made for example by GRPC spawning two threads to serve two concurrent requests. The main purpose of NodeState.java class was to make it easier to synchronize access of Pastry node to its routing structures. The class therefore uses single ReentrantLock upon each operation on the routing structures.

During the implementation, an attempt to outsource the tedious locking and checking for presence of each individual node references within the routing structures was made: plain Java's ArrayLists were replaced by `Collections.synchronizedSortedSet()`. As the name may suggest, the collection promises unique sorted items with thread-safe access by default. Our expectation was to sort by metric that can be chosen and define uniqueness of node by ip an port, therefore not keeping two nodes of same ip and port in the set. To use the collection however, required implementing Comparable interface for node references. Usage of this collection was in the end rejected for two main reasons:

1. NeighborSet sorts nodes according to the metric, whereas LeafSet sorts

nodes according to numerical distance. This cannot be achieved using single Comparable.

2. Comparable interface requires the ordering to be consistent with *equals()* method. This would be violated by our requirement to uniquely identify node by ip and port combination.

5.1.2 GRPC server/client

Also similar to the Chord protocol implementation, the primary class `PastryNode.java` incorporates both the GRPC client and server functionalities. There is a couple of important GRPC services implemented:

- join()
- forward()
- notifyExistence()

As the name suggests, *join* service is used when new node is joining the network. The bootstrap node A routes the request to the node Z which is numerically closest to the joining node. Nodes along the way of join call fill the response with their node state. Joining node will then use these to initialize its own node state.

Note that GRPC requires a pair of request and response for a successful RPC. When joining occurs, the initial request is propagated to the node Z and only then the response is created, which is again propagated back to the initiating node. Node states from each intermediate nodes are inserted into the response on the way back.

Forward RPC is identical in terms of propagating request/response objects through the network. Difference however being, that forward service is meant for routing values, not nodes. The destination node Z therefore stores the value, since it has the closest id to the value, and intermediate nodes only forward the response back without any ”enrichment” of the response.

Finally *notifyExistence* serves as presence notifier after joining the network. Node notifies whole network about its presence. It does so by requesting known nodes (discovered during join) for their known nodes until there are nodes that have not yet been notified. Notified node may decide to split its local data and send it to the requestor if there are keys that are closer to the requestor then the notified node.

5.1.3 Distance calculator

By default, Pastry node works with no distance metric set. Implementation however offers the user to make his own distance metric by implementing the `DistanceCalculator` interface. The implementation comes with `CoordinateDistanceCalculator` which when used prefers routing table candidates that are closer according to X/Y coordinates. That in turn makes Pastry choose route that is ”good” with respect to the metric. User can also experiment with other included distance calculators.

5.2 Leave

Anytime a node decides to leave the network, it transfers responsibility over keys to its closest leafs. One can observe that the closest upLeaf and closest downLeafs are indeed valid nodes for responsibility transfer since these nodes are the closest in terms of id to the departing one and therefore will also be the closest nodes in terms of id to the keys it stored.

5.3 Fail recovery

Fail detection was implemented as mentioned in the paper, where node is considered failed when its immediate neighbors can no longer communicate with the node. Stabilization thread is therefore spawned every 5 seconds (default value) that checks the liveness of each neighbor. If neighbor is unavailable, it is unregistered from the node state and new appropriate node is found.

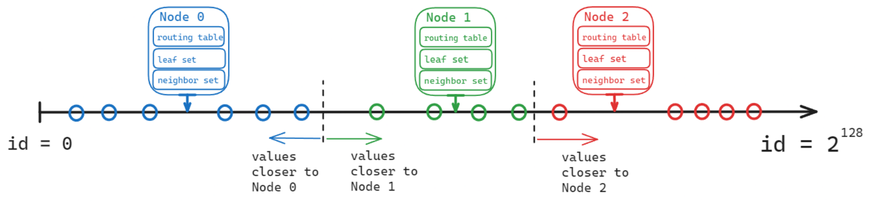
5.4 Future work

5.4.1 Whether to wraparound or not

During our work a seemingly important discussion had arisen when comparing Chord and Pastry: whether Pastry ”does or doesn’t do wraparound”. Following sentences aim to shed light on this question.

The question most probably stems from Chord’s description of its network topology. As even the authors state [Sto01] Chord would probably work even without the fingertable, just with predecessor and successor references, although not very efficiently. Let us therefore suppose a very trivial Chord network in which no node keeps a fingertable. In such a network, when placing an item (station or value) into the network, we walk *around* the network, eventually finding the right spot. During the walk, we may encounter a station that has the highest id among all the stations in the network. Given the case that we still have not found the right place to insert our item, we will continue our walk toward the next station which will be a station that has the lowest id among all the stations. At this moment we have effectively did a **wraparound**.

With the notion of wraparound defined, let’s ask another question: what does a wraparound mean, in a a mesh-like network? Note that in Pastry there is no notion of successor nor predecessor. Does wraparound in Pastry mean a reroute from highest id node to lowest id node? What about from highest to the second lowest? And at what point does wraparound become a plain reroute? Answers to these questions should illustrate the pointlessness of such questions. Whether Pastry does or does not do a wraparound probably matters not. With this logic, we come to a conclusion that the wraparound



Rough sketch of Pastry system. Line represents the id number line. Circles on the line represent values stored in Pastry DHT. Color of a node signifies the owner of the value. The owner of a given value is the node whose id is numerically closest to the value's id. Source: author

Figure 5.1: Pastry identifier space

is most likely a question of semantics. Nonetheless a question of semantics which should be made clear when explaining to a colleague or student.

Lastly, one should also observe, that there is no wraparound for storing values. Not even in the "from highest to lowest id" sense. Values are simply stored on the numerically closest nodes. Therefore it makes no sense for highest id value to be stored on lowest id node or vice versa.

5.4.2 Visual representation of Pastry system

With the wraparound question 'answered', we would like to propose a sketch of the Pastry system focused on positions of stored values relative to their owning machines. Contrary to plenty drawings of Chord system in which the ring represents id number line but also the true connection between neighboring nodes, the line in our sketch represents *only* the id number line. Although this sketch is similar to a disconnected Chord drawing, we are of the opinion that this can still be used to facilitate the understanding of Pastry protocol. Moreover potential future works with ambitions of visualizing the Pastry system would still benefit from at least some representation of values and nodes within the system. The figure 5.1 shows proposed sketch of the system.

Future work aimed at visualizing the Pastry protocol may also benefit from the implementation presented here. We conclude that a single extra node may serve as a central visualizing unit which could draw out each RPC made by a node in the network, similarly as was described in previous Chord chapter. Visualizing Pastry however may be more complex because of the intricacies presented by node's routing structures. Therefore, extra steps should be taken not just to visualize the calls but also changes in the routing structures.

Chapter 6
Kademlia

The process of implementing the Kademlia DHT followed a ”bottom-up” approach: `KBucket.java` was created as the first testable unit of code and RoutingTable containing K number of buckets followed. Both classes contained a good amount of complex yet testable mechanisms (compared to Pastry’s routing table) for us to design some simple unit tests. These unit tests may actually provide some basic understanding of Kademlia’s confusing routing table.

Consistently with previous implementations, `KademliaNode.java` contains the logical unit of Kademlia network node. It was again split into server-client GRPC sides within a single class.

6.1 KBucket

`KBucket.java` is a simple class containing ArrayDeque of references to other Kademlia nodes. ArrayDeque is used to provide fast access to the first and the last element in the bucket. Single KBucket preserves most recently seen order of nodes by inserting them to the tail/end of the ArrayDeque as they come. In case an already contained node is encountered and is to be inserted, it is removed from the ArrayDeque and reinserted to the tail/end of the ArrayDeque. The least recently seen nodes are dropped if bucket is full.

Since our implementation treats routing table as an array, rather than binary tree, all 160 KBuckets are initialized upon KademliaNode creation as well as scheduled for refresh at the same time (will be explained in greater detail later).

KBucket also offers a *toStream()* method which converts the ArrayDeque into a Java stream. Java functional stream interface comes very handy when selecting the K closest nodes to a given ID.

Listing 6.1: Find K closest using the java stream interface

```
public List<NodeReference> findKClosest(BigInteger targetId) {
    return buckets.stream()
            .filter(b -> !b.isEmpty())
            .flatMap(KBucket::toStream)
            .sorted(Comparator.comparing(node ->
                targetId.xor(node.getId())))
            .limit(K_PARAMETER)
            .collect(Collectors.toList());
}
```

6.2 RoutingTable

The RoutingTable.java represents the Kademlia binary tree. It does so by storing list of 160 KBuckets, each bucket indexed by the position of the highest non-zero bit of XOR distance between owner of the table and the target id. Observe that this corresponds to the correct bucket since the first non-zero bit of XOR distance marks the position of the first prefix mismatch.

6.2.1 Find K closest

While the structure of the table and the indexing of buckets seems elegant, finding the K closest nodes in terms of XOR distance proves to be less so. To return a whole bucket given by XOR distance is trivial. There are however cases when the bucket is not full and we have to scan the whole routing table since neighboring bucket don't necessarily contain closest nodes [Maz17b, Mul24].

6.2.2 Array versus Binary tree

Note that this array-like routing table implementation does not correspond to binary tree implemented as an array. Whereas binary tree may more closely resemble the true protocol described in the final versions of Kademlia paper, array-like structure was actually used in the pre-print version for sketches of proof [Maz17a]. We therefore conclude that array-like structure does not violate the correctness of protocol.

It should be noted, that actual runtimes will differ between both structures. The decision to use array-like routing table rather than binary tree version came from the fact that it is easier to implement as well as to use. Easier implementation comes from the fact that there is no need for bucket splitting when dealing with highly unbalanced trees. Increased ease of use comes from the fact, that when we try to find K closest and given bucket is not full, we have to scan the whole routing table. This would mean traversing the whole tree. ArrayList however offers quite elegant solution, especially when we use java streams as in listing 6.1.

6.3 KademliaNode

The `KademliaNode.java` represents a single node in the network. Aside from the central *node lookup* procedure, it provides the usual DHT API methods. Each of the DHT API method is implemented using asynchronous GRPC stub on top of node lookup procedure. It initially finds K closest nodes using the *node lookup* followed by K number of asynchronous put/get/delete requests to these nodes. GRPC's asynchronous stub provides *onNext()* and *onError()* callback methods. These were implemented to decrement `java.util.concurrent.CountDownLatch` upon response receival. The latch will then block the main thread until it reaches zero, effectively synchronizing threads until all K responses arrive back.

6.3.1 Node lookup

Since the explanation of node lookup in the original Kademlia is not very clear, we decided to implement a helper class `Shortlist.java` to reduce the cognitive load. Shortlist represents the list of all responses from FIND_-NODE RPC. Recall that response to FIND_NODE RPC consists of K closest nodes from recipients perspective. Shortlist keeps track of already queried, yet unqueried and offline nodes. Node lookup itself was implemented to asynchronously query *alpha* unqueried nodes while Shortlist contains unqueried nodes. Once Shortlist contains only queried nodes, Shortlist is prompted to return K queried online nodes closest to given ID.

Listing 6.2: Node lookup procedure using Shortlist structure

```java
private List<NodeReference> nodeLookup(BigInteger id) {
    Shortlist SL = new Shortlist(routingTable.findAlphaClosest(id));
    while (SL.hasUnqueried()) {
        // a round of alpha FIND_NODE RPCs
        multicastFindNode(SL, id, joiningNode);
    }
    SL.getOffline().forEach(routingTable::remove);
    return SL.getKBestQueried(id, K_PARAMETER);
}
```

6.4 Scheduling

Compared to Pastry and Chord, Kademlia required more sophisticated approach when it comes to scheduling tasks. We therefore decided to use `java.util.concurrent.ScheduledThreadPoolExecutor` to handle scheduled tasks. These tasks can be categorized into three groups:

- RepublishTask - when a node is prompted by user to insert a value into the DHT, it regularly republishes it onto K closest

- ExpireTask - when a node is prompted by other node to store a value, it will expire the value after specified duration if it is not republished
- RefreshTask - this task is initially scheduled on all 160 buckets, a node will *refresh* otherwise unaccessed bucket after specified duration

To store a value, user picks a node and calls a *put* method on it. This node now becomes *original publisher* of the value and schedules the republish procedure in intervals of specified duration.

ExpireTask occurs if a node that previously received store request for a particular value does not receive it again. Node removes the value from its local data after specified duration.

Node will *refresh* a given bucket if there was no traffic on the bucket after specified duration. RefreshTask generates a random id from within the bucket and prompts a node lookup on it [May02]. This will refresh the entries within the bucket with the responses that come back, potentially removing unresponsive nodes.

6.5 Locking

As well as with Pastry and Chord, Kademlia also employs some form of locking to prevent concurrent access. There are two `ReentrantLocks`: one in the routing table preventing concurrent editing of any buckets, second in the node class, preventing concurrent access to local data and task executor. Since ReentrantLock requires lengthy try-lock blocks, we decided to make a 'lockWrapper' method. This method accepts a Runnable interface, which in java can be a function. Lockwrapper itself calls puts runnable.run() in between the try-lock blocks, effectively locking all structures used in the runnable. This mechanism was later reused to also work on *get()* method of the DHT interface to lock a java Supplier. Note that Supplier is a Runnable that returns a value.

Listing 6.3: Example of lockWrapper use

```java
private void lockWrapper(Runnable action) {
    lock.lock();
    try {
        action.run();
    } finally {
        lock.unlock();
    }
}

public void delete(String key) {
    BigInteger keyHash = getId(key);
    if(routingTable.getSize() == 0) {
        lockWrapper(() -> localData.remove(keyHash));
    } else {
    ...
    }
}
```

6.6 Parametrization

Aside from *alpha* and K parameters, our implementation also lets the user to specify the length of node id. Note that configuring this affects the number of buckets each node has in its routing table. Durations of all scheduled tasks is configurable as well, since the default values presented in the original paper don't make much sense for educational use.

6.7 Future work

Future works with ambitions of vizualizing Kademlia protocol based on our implementation can again take an advantage from a single central node. This node would have to be specified globally, the same way as K, *alpha* and other parameters are. Each peer node would then be obligated to notify the central node about an RPC it has made. Central node would then again draw out each RPC that the peer sent.

Chapter 7
The DHT Library

The final part of our implementation comprised of designing a wrapper library around the three implementations. This library offers common API of Chord, Pastry and Kademlia under a single and easy to use interface.

Designed library contains simple interface class `DHTNodeInterface.java`. It requires the implementing class to override the basic put, get and delete functions as well as couple more for initialization, join and leave. The second class `DHTNode.java` represents abstract node of a chosen implementation. Not only can DHTNode act as Chord, Pastry or Kademlia node, it also offers the user the ability to statically customize parameters of each type of DHT. After statically defining custom settings or keeping the defaults, user can then initialize a node of the desired type by inserting type enum into DHTNode constructor.

Finally, the library itself contains unit tests as well. These can give the user additional hints as to how to use the library.

7.1 How to use it

7.1.1 Create a bootstrap node

In order to create for example a bootstrapping Chord node, user will input the Chord node type[1] and an ip & port combination to the DHTNode constructor. This in itself however won't create a DHT instance. To create one, the library requires user to call *init()* function on a given bootstrap node. This call will then start a GRPC server on the specified port, in effect initializing a single-node DHT of specified type. This is a minimal working DHT. The user can then create more nodes and make them join the bootstrap node. Note that all nodes in a single DHT must have the same type!

[1]Type of node is an enum.

Listing 7.1: How to use the DHT

```
DHTNode bootstrap = new DHTNode(DHTType.Chord, "localhost", 8000);
bootstrap.init();

DHTNode joiner = new DHTNode(DHTType.Chord, "localhost", 8001);
joiner.join(bootstrap.getIp(), bootstrap.getPort());
```

7.1.2 Join DHT instance using the bootstrap node

To join an existing DHT, user has to know a contact within a network. Since we have an already running one, we can specify it's ip & port combination in the *join()* function and connect via this node. Observe that even though the listing 7.1 runs the node on the local environment, user is able to specify arbitrary remote ip address.

7.1.3 Customizing the network

So far we have shown how to create a Chord DHT with no configuration. Nodes initialized without prior settings use a default configuration. Chord nodes are run by default with m parameter set to 16 and *stabilizeInterval* set to 2 seconds. These parameters can be customized however. To do so, one only has to call static method *setCustomChordSettings()* which takes both arguments as input. The m parameter determines the node id space. There are 2^m ids in a given Chord instance. The *stabilizeInterval* is the number of **milliseconds** after which a node runs stabilize routine. This routine checks whether node's successor is online and updates a random finger.

Pastry and Kademlia nodes can be constructed almost identically to the code presented in listing 7.1. The only difference is the type enum. Pastry and Kademlia also offer the user to customize their id space and other parameters. Pastry node allows for customization of three parameters:

- idBase - allowed values are 4 and 16
- leafSize - 8, 16 or 32
- distanceCalculator - accepts classes of DistanceCalculator interface

User can customize the base of Pastry node ids to be in base 4 or 16 by setting the *idBase* argument. Recall that the base of Pastry ids directly influences the size of routing table row. LeafSize denotes the size of leafSet. Setting the value as 16 for example will reserve 8 slots for upleafs and 8 slots for downleafs.

LeafSize also denotes the L parameter from the original paper [Row01]. The user should therefore only expect eventual delivery under the assumption, that no $\frac{leafSize}{2}$ nodes with adjacent ids fail at the same time!

Finally, the *distanceCalculator* is an interface which aims to define a metric that should be biased during routing. The library itself contains pre-implemented metrics which can be utilized. There is for example `CoordinateDistanceCalculator` which biases euclidean distance[2]. The other noteworthy calculator is the `ZeroDistanceCalculator`, which always returns zero, therefore equalizing all distances. This in effect negates the metric routing bias. The user is free to implement his own metric by implementing the DistanceCalculator interface.

Kademlia's settings offer double the amount of Pastry parameters. There are 6 parameters in total, three of which are duration intervals:

- *k* - accepts values in range <1, 20>
- alpha - accepts values in range <1, 10>
- idLength - accepts values in range <4, 160>
- refreshInterval - no bounds
- republishInterval - no bounds
- expireInterval - no bounds

The *k* and alpha arguments correspond to the *k* and alpha arguments from the original paper. *k* sets the size of *k*-buckets whereas alpha is the concurrency parameter.

The idLength argument represents the number of binary bits of id. The original Kademlia paper talks about 160-bit length id [May02]. Since this is very unwieldy, the length was made configurable during the implementation. It stayed this way because it is very hard to identify nodes of such lengths in the debug console.

The refreshInterval parameter denotes the duration after which an otherwise unaccessed bucket will be refreshed. Refresh means picking a random node from within a bucket and looking it up in order to update contacts in the given bucket [May02].

RepublishInterval corresponds to the duration after which the publisher republishes his keys. ExpireInterval denotes the duration after which a key is removed from a given node.

7.1.4 REST API

A simple REST API was designed in order to make node deployment and testing easier. The DHT library contains `Main.java` class which can be run from commandline or IDE. The class takes two integer arguments: DHT and REST endpoint ports.

To run a minimal working two-node DHT instance, user will start two instances of this class, each instance with two ports specified. Suppose the user runs the first instance with DHT port set to 8080 and REST port set to 8081. Further suppose the second instance is started similarly with DHT port set to 8090 and REST port set to 8091.

[2]The coherency of DHT interface goes against deviations which different nodes of different DHT types have. One such deviation is setting XY coordinates on each pastry node. Since only Pastry node constructor deviated from others, it was decided to set random XY coordinates to such nodes when using the library. Curious user can however bypass the DHT library and use the Pastry XY coordinate constructor directly.

Once both instances are started, nodes per each instance are running but are not initialized. In order to initialize a node, user will send GET request to it's REST endpoint. In this case, it would be on `http://localhost:8081/init` and similarly for the second node at `8091`. After both nodes are started, the user can prompt a node to connect to a running node by sending a GET request to `http://localhost:8091/join/localhost/8080`. This request will make the second node join the first. The REST API allows the user to prompt the node to do the *standard* DHT node operations. These include for example value storage, retrieval, deletion and many other. Refer to the Table 7.1 for an overview of all operations which can be prompted on a node's REST port.

This REST API also allowed for validation of the implementation in a virtual environment. For each three of the implementations tested (Chord, Pastry, Kademlia), there were six nodes deployed with eight key-value pairs stored. Value storage was then validated through get requests sent on various nodes in the network. Finally, the node leave, shutdown and failure simulation was validated in the test scope as well.

Table 7.1: DHT Node's REST endpoint and its prompted operations.

URL	Operation
http://{nodeIP}:{restPort}/put/{key}/{val}	Store a key-value pair
http://{nodeIP}:{restPort}/getLocalData	Retrieve all stored key-value pairs on the node
http://{nodeIP}:{restPort}/get/{key}	Retrieve the value for the given key
http://{nodeIP}:{restPort}/delete/{key}	Delete the key-value pair for the given key
http://{nodeIP}:{restPort}/shutdown	Shut down the node
http://{nodeIP}:{restPort}/fail	Simulate a node failure
http://{nodeIP}:{restPort}/leave	Remove the node from the network

Chapter 8
Results and Conclusions

8.1 Implementation summary

After designing and implementing three different types of DHT, we observed a certain pattern which the process followed. The process usually involves two main steps. First is defining the routing structure that the node of given DHT type uses. In our case this was the finger table, node state and list of buckets. Following the routing structure's design, the *heart* of the protocol can be implemented - the actual routing algorithm.

The routing procedure seems to be the most challenging one. Future developers are highly advised to construct many test cases to validate their implementation. Our experience proved, that seemingly correct routing procedure may not actually properly converge in a network of many nodes.

As soon as both, the routing structure and algorithm are designed, all three main methods from the DHT usually only build on top of these. All three methods prompt the routing procedure to return some number of *nodes of interest*. Recall that these nodes are characterized by some form of closeness to a given key. As soon as the routing procedure returns, we can directly invoke *put(), get()* or *delete()* RPCs on these nodes to store, retrieve or delete a given value.

Although routing procedure and routing structure are two main points of DHT implementation, there are also other details as well. The next most important perhaps is the synchronization of routing structure and other local data stored on the node. To make the implementation simpler, we used ReentrantLock on each local data access. This surely leads to a bottleneck when processing, but also trivially ensures there is no concurrent editing.

8.2 Assessment

Author believes that the presented implementations have been designed and tested successfully. Furthermore, each implementation mirrors the described one quite precisely, although there are some details that have been missed out. In particular, these is no successor list in Chord, which allows for more effective fail recovery or caching of nodes in Kademlia. Lastly, Kademlia diverges from original binary tree structure of routing table by using an array of buckets.

During the implementation, there also seemed to be a difficulty in scaling out each implementation in local environment. After extensive testing, we came to a conclusion that this is probably not caused by local resource exhaustion, but rather some sort of race condition. This intuition comes from the fact, that the designed library crashes when greater number of nodes are connected but seems to run fine when we introduce a little delay between the presence-notifying calls after a new node joins a network.

The author is furthermore confident of correctness since there is a great amount of unit and integration tests designed in each of the three implementations. This however does not guarantee correctness on itself. Each DHT most probably contains various edge cases that have not been covered. Furthermore, implementation presented here has never been deployed nor stress-tested.

Nevertheless, the author believes the main goal of this thesis, the designing and developing a DHT for educational purposes, had been achieved successfully. On top of that, the library also provides various logging levels which the user can configure for more fine grained event notification. This may provide the user with better understanding the hidden mechanisms of the system. On top of that the author also strived for clear code documentation, plenty of times even quoting the authors of original papers in the code itself. The author would also like to encourage any future users of the DHT library to set break points in the code itself and step through each of the mechanisms implemented, or to set custom logging messages through out the code.

8.3 Future work

Future works may directly build on top of the one presented here. These works may for example extend the current project with visual and animated elements representing the identifier space as well as processing of RPCs sent between the stations. As discussed in previous chapters, we propose a simple solution, in which one extra central node connects to the network and orders other nodes to notify the extra node every time it sends an RPC.

We imagine that there are other works which may benefit from the presented project as well. These may include a distributed file-sharing system for example. Recall that the implementations presented, stores simple string values on the nodes. There is however a possibility to extend this implementation to also store files or data blobs in general.

8.4 Conclusion

As with many complex systems with plenty of moving parts, distributed systems may be overwhelming at a first glance. Nevertheless, even the most complex systems are usually made of simple atomic building blocks which are just built on top of each other. To explain and illustrate the building blocks first and only then continue with advanced topics seems like a natural way one can learn a given subject. This thesis was also presented in such approachable way. It introduced the reader to a concept of consistent hashing, continuing with metric heuristic and finishing of with an array of k buckets. The thesis not only introduced the reader to the topic in theoretical way but also showed how to implement such system from scratch. Author believes that the work at hand will greatly contribute to academic and software developer community by educating future readers or users of results presented here.

Appendices

Appendix A
Bibliography

[Bas16] Basho, *Technical overview of riak kv enterprise*, Tech. report, Basho, 2016.

[Con24] Wikipedia Contributors, *grpc*, `https://en.wikipedia.org/wiki/GRPC`, 2024, Accessed: 2024-02-23.

[Cou11] George Coulouris, *Distributed systems: Concepts and design*, Addison-Wesley, 2011.

[DeC07] Giuseppe DeCandia, *Dynamo: amazon's highly available key-value store*, ACM SIGOPS Operating Systems Review (2007).

[Elb15] Waleed Elbreiki, *A comparative study of chord and pastry for the name resolution system implementation in information centric networks*, NETAPPS2015, 2015.

[gA24] gRPC Authors, *Java basics*, 2024, Accessed: 2024-02-23.

[Gho06] Ali Ghodsi, *Distributed k-ary system: Algorithms for distributed hash tables*, IEEE transactions on parallel and distributed systems (2006).

[Kar97] David Karger, *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web*, ACM Transactions on Computer Systems (1997).

[May02] Petar Maymounkov, *Kademlia: A peer-to-peer information system based on the xor metric*, International Conference on Distributed Computing Systems, 2002.

[Maz17a] David Mazières, *Implementing find node on torrent kademlia routing table*, `https://stackoverflow.com/questions/30654398/implementing-find-node-on-torrent-kademlia-routing-table`, 2017, Accessed: 2024-04-21.

[Maz17b] ______, *Is there a clearly documented consensus in plain english that describes maymounkov's kademlia?*, https://stackoverflow.com/questions/47507317/is-there-a-clearly-documented-consensus-in-plain-english-that-describes-maymounk/47517428#47517428, 2017, Accessed: 2024-04-21.

[Mul24] Brian Muller, *K closest neighbours, issue 28*, https://github.com/bmuller/kademlia/issues/28, 2024, Accessed: 2024-04-21.

[Rou22] Tim Roughgarden, *Cs168: Lecture #1: Introduction and consistent hashing*, 2022.

[Row01] Antony Rowstron, *A scalable and fault-tolerant router for large-scale peer-to-peer networks*, Computer networks (2001).

[Ste05] Ralf Steinmetz, *Peer-to-peer systems and applications*, Springer, 2005.

[Sto01] Ion Stoica, *Chord: A scalable peer-to-peer lookup service*, ACM SIGOPS Operating Systems Review (2001).

[Tan07] Andrew S. Tanenbaum, *Distributed systems: Principles and paradigms*, John Wiley & Sons, 2007.

[Zha13] Hong Zhang, *Distributed hash table: Theory, platforms, and applications*, Springer, 2013.

Appendix B

Attachment content

```
attachment/
  DHT/src/
    main/
      java/
        chord/ ............ Chord DHT source code
        pastry/ ........... Pastry DHT source code
        kademlia/ ......... Kademlia DHT source code
        dht/ .............. The DHT library source code
        proto/ ............ Proto compiler generated files
      resources/
        chord.proto ....... Chord gRPC service definition
        pastry.proto ...... Pastry gRPC service definition
        kademlia.proto .... Kademlia gRPC service definition
        dht.properties .... DHT custom settings file
        logback.xml ....... Logging level configuration
    test/java/
      chord/ .............. Chord DHT tests
      pastry/ ............. Pastry DHT tests
      kademlia/ ........... Kademlia DHT tests
      dht/ ................ The DHT library tests
  assignment.pdf .......... Official thesis specification
  chord.pdf ............... The original Chord paper
  pastry.pdf .............. The original Pastry paper
  kademlia.pdf ............ The original Kademlia paper
```