Design, Implementation and Evaluation of Merging Mechanisms for large-scale and dynamically partitioned Networks

Master Thesis

by

Tobias Amft

born in
Tönisvorst

submitted to

Technology of Social Networks Lab
Jun.-Prof. Dr.-Ing. Kalman Graffi
Heinrich-Heine-Universität Düsseldorf

March 2014

Supervisor:
Jun.-Prof. Dr.-Ing. Kalman Graffi

Abstract

Structured peer-to-peer overlays define a robust and scalable alternative to the centralized server architecture. Hence those overlays can be used by various applications, in order to offer secure communication and prevention against espionage or censorship attempts from governments or cyber criminals. Political revolutions in recent years have shown that it is inevitable for peer-to-peer overlays to be aware of Internet isolations, due to governmental arbitrariness. In this thesis we characterize network partitioning events and summarize criteria, which have to be focused on, to unify split ring-based overlays, as soon as connectivity is established again between two or more network partitions. We then present the Ring Reunion Algorithm, a novel merging algorithm, which automatically detects network partitions and initializes merging operations. The algorithm includes methods to start parallelized merger instances autonomously and to reduce the amount of messages during the merger process. In stable overlay states, the Ring Reunion Algorithm terminates quickly to limit unnecessary message overhead. The Ring Reunion Algorithm, in its simple and parallelized version, is compared to Chord-Zip, two variations of the Ring Unification Algorithm and the basic Chord protocol in a detailed evaluation, which bases on realistic simulations of different scenarios. The evaluation results prove that the Ring Reunion Algorithm is able to merge multiple isolated ring-based overlays effortlessly and in parallel. Even under heavy churn our approach merges separated networks quickly and reliably. Meanwhile, instead of depending on the quantity of participating nodes, the message complexity depends on the number of constructs that form the overlay which is currently supposed to be merged. Therefore message overhead, caused by unnecessary merger attempts, is almost completely avoided in an united overlay. Finally, considering our simulations, we give suggestions on how the Ring Reunion Algorithm should be configured to merge different scenarios reliably, fast and with low operational costs and message overhead.

Acknowledgments

This thesis is dedicated to my mum, Antje Amft, who said to me once upon a time: do something (good) with your life.

Long story short, I wish to thank all people who supported me throughout all the years, especially: Vanessa Amft, Maren Amft, Dieter Amft, Angkhan Choeichuen, Andrej and Eddy Henkel, and many others.

I would like to give special thanks to Jun.-Prof. Dr.-Ing. Kalman Graffi, who supervised this thesis and gave a lot of useful tips.

Thank You!

Contents

List of Figures xi

Listings xiii

1 Introduction 1

1.1 Motivation . . . 1
1.2 Characteristics of Network Partitioning . . . 3
1.2.1 Separation of Single Nodes . . . 3
1.2.2 Separation of Whole Groups . . . 4
1.2.3 Requirements for Merging Algorithms . . . 6
1.3 Outline . . . 8

2 Related Work 9

2.1 Background . . . 9
2.1.1 Chord . . . 10
2.1.2 Chord Variants . . . 12
2.2 Tale of Two Networks . . . 13
2.2.1 Case-Study Chord . . . 14
2.2.2 Case-Study P-Grid . . . 15
2.3 Analysis of P-Grid . . . 16
2.4 Chord-Zip . . . 18
2.4.1 Summary of Chord-Zip . . . 18
2.4.2 Discovery of Contact Nodes . . . 19
2.4.3 Merging . . . 19
2.4.4 Parallelization . . . 21
2.4.5 Termination . . . 21
2.4.6 Robustness . . . 22
2.5 Ring Unification Algorithm . . . 22
2.5.1 Summary of the Ring Unification Algorithm . . . 23
2.5.2 Discovery of Contact Nodes . . . 24
2.5.3 Merging . . . 24

2.5.4 Parallelization . . . 26
2.5.5 Termination . . . 26
2.5.6 Robustness . . . 26

3 Ring Reunion Algorithm **29**
3.1 Design of the Ring Reunion Algorithm . . . 30
3.1.1 Discovery of Contact Nodes . . . 30
3.1.2 Merging . . . 31
3.1.3 Probability . . . 32
3.1.4 Parallelization . . . 33
3.1.5 Termination . . . 35
3.1.6 Robustness . . . 36
3.2 Performance Analysis . . . 37
3.2.1 Time Complexity . . . 37
3.2.2 Message Complexity . . . 38
3.2.3 Lookups During Merger . . . 39

4 Implementation **43**
4.1 PeerfactSim.KOM . . . 43
4.1.1 Components . . . 44
4.1.2 Simulations . . . 45
4.2 Implementation . . . 46
4.2.1 Contact List . . . 46
4.2.2 Merger . . . 47
4.2.3 Isolation Model . . . 48
4.3 Metrics . . . 48
4.3.1 Number of Constructs . . . 49
4.3.2 Correct Pointers . . . 49

5 Evaluation **51**
5.1 Scenario Setups . . . 52
5.1.1 Merging Manually . . . 52
5.1.2 Merging Automatically . . . 54
5.2 Simulation Results . . . 57
5.2.1 A.1: Merging Two Rings . . . 58
5.2.2 A.2: Merging Three Rings . . . 58
5.2.3 A.3: Merging Five Rings . . . 60
5.2.4 B.1 Comparison of Performance . . . 61
5.2.5 C.1: Study of Passive List . . . 62

5.2.6 C.2: Study of Active Contact List 62
5.2.7 C.3-5: Study of Probability 64
5.2.8 D.1: Complex and Realistic Scenario 66
5.2.9 E.1-2: Parameter Studies and Churn 67

6 Conclusion **71**

Bibliography **73**

List of Figures

1.1 Ring is corrupted during isolation. 5
1.2 Groups are formed after one region has been isolated. 5

2.1 Hash Table and Distributed Hash Table. 10
2.2 Chord Lookup: path of query for key 58, started at node 14. 11
2.3 Example of a P-Grid instance. 15
2.4 State of P-Grid instances after merger. 17
2.5 Example for partitioning in P-Grid instance. 18
2.6 Node *n* skips its own successor in Chord-Zip. 20
2.7 Messages in the Chord-Zip protocol. 21

3.1 Example: groups in the same region can be merged during isolation. 30
3.2 Ring Reunion Algorithm selects suitable alternative successor. 32
3.3 Affected nodes, in case the distribution algorithm is used. 32
3.4 Verification of Ring Reunion Algorithm (1): two nodes try to merge ring at the same time. 34
3.5 Verification of Ring Reunion Algorithm (2): two nodes merge ring at the same time. . 35
3.6 Example: one node tries to merge own ring. 36
3.7 Ring Unification Algorithm operates in both directions: clockwise and anti-clockwise. 38
3.8 Lookup during merger after routing information has been combined. 41
3.9 Successful lookup during merger. 41

4.1 Possible overlay constructs which arise during a merging process. 49

5.1 Scenario in which five groups are merged. 54
5.2 A.1: Merging of two networks. 59
5.3 A.2: Merging of three networks. 59
5.4 A.3: Merging of five networks. 60
5.5 A.3: Closeup on merging of five rings. 61
5.6 B.1: Comparison of Ring Reunion Algorithm and Ring Unification Algorithm. 61
5.7 C.1: Passive list used to find alive contact nodes. 63

5.8 C.2: Active contact list is iterated periodically to find further nodes. No probability is used to reduce message overhead. 63
5.9 C.3: Gossip-based Ring Unification Algorithm, overlay constructs start α merger instances every 4 minutes on average. 64
5.10 C.4: Simple Ring Reunion Algorithm in combination with alive contact list. 65
5.11 C.5: Ring Reunion with 4 parallel instances in combination with alive contact list. . . 65
5.12 D.1: Complex realistic scenario with usage of alive contact list and 10 started merger instances per construct. 66
5.13 E.1: Ring Reunion Algorithm with parallel instances, $\alpha \in \{10, 100\}$. 68
5.14 E.2: Ring Reunion Algorithm with parallel instances and various values for interval parameter, $\alpha = 10$. 69

Listings

2.1 Chord-Zip Algorithm. 20
2.2 The Simple Ring Unification Algorithm. 25
2.3 Gossip-based Ring Unification Algorithm. 25

3.1 Ring Reunion Algorithm. 31
3.2 Ring Reunion Algorithm with parallelization. 33
3.3 Basic Chord methods. 40
3.4 Chord's methods for stabilization. 40

4.1 Example of a XML configuration file. 45

Chapter 1

Introduction

Surely, speech is the most outstanding ability of mankind. It contributes to communication, sympathy and to disclosure of information among people all over the world. Precisely this is the reason why freedom of speech and information dissemination should be seen as fundamental right of every human being and has to be protected as such. Unfortunately, still many countries exist, in which freedom of speech is suppressed by the respective government. According to the *Amnesty International Report 2013*, the right to speak freely was suppressed in 101 countries in 2012 [1].

1.1 Motivation

The important role of media in the context of information dissemination can not be denied. Already in the year 1514, the German monk and seminal figure of the Protestant Reformation, named Martin Luther, benefits from the progress in printed media, which has been driven forward by Gutenberg's work on his printing press in the fifteenth century. Without the ability to print books in large quantities with low costs it had not been possible to publish Luther's translation of the Bible in the way it was done. Furthermore Gutenberg's printing press and Luther's Bible contributed to form and standardize the German language in many ways. Probably the most important medium nowadays is the Internet, which provides the ability to access information all over the Earth (and beyond). Therefore it is not surprising that governments of many countries attempt to control parts of the Internet to control its citizens likewise.

With the Golden Shield Project for example, the Chinese government "filters Internet traffic in and out of the country" [2]. Further, China uses different techniques to suppress specific information exchanges purposefully. Beside the usage of DNS poisoning, traffic to certain IP addresses is directly blocked and discarded. Another technique used by the Great Firewall of China is based upon intrusion detection system (IDS) mechanisms. Internet traffic is searched for keywords which are considered

to be suspicious, e.g. if they concern political groups or activities that are not tolerated by the regime. Connections which are tested positive for such keywords are forced to be closed by the Chinese firewall [3].

Another drastic and current example of suppressing freedom of speech and the dissemination of information is the behavior of the Egypt government during the *Arab Spring* in early 2011. The uniqueness of the protests during the Arab Spring is characterized by the enormous usage of social media and familiar Internet platforms like Facebook, Twitter and YouTube, which mainly have been used to organize demonstrations, to share informational content or simply to criticize the government. Furthermore, social media was used to communicate with Western countries and to exchange information with people living in other countries that were involved in the Arab Spring movement [4]. According to the *Institute of World Economy and International Relations (IMEMO)*, Facebook even outmatched the Arabic TV channel Al Jazeera "in at least the speed of news dissemination" [5].

As a response to the revolutionary movements in North Africa, the Egyptian government instructed mobile operators and Internet service providers to suspend their services. As a result most parts of the Internet had been cut off. Only few governments had done this before: Nepal cut off Internet access entirely in 2005, as did Myanmar two years later in 2007 [6]. From 27th of January 2011 until 2nd of February 2011, approximately 93% of Egyptian networks had been unreachable [7]. First hints that the Internet might break down had been given one day before as Twitter and Facebook suddenly had been discovered to be unreachable [8]. Nevertheless, the entire disruption could not have been foreseen or prevented.

The cases of Nepal, Myanmar or Egypt are rare examples of network partitioning that could be repeated any time. They show clearly that governments of countries with simple Internet infrastructure are capable of stopping Internet traffic almost entirely. In such a case most Internet services, and therefore social media, are not reachable, because the provider's servers are mostly located in the USA (like Twitter, Facebook, YouTube, Yahoo, etc.). Another problem arises with traditional client-server approaches: a lack of privacy and trust. Centralized servers constitute a single point of failure and are convenient to intercept and manipulate information directly. The NSA affair is a current evidence that Internet services like Facebook, Yahoo, Twitter and so forth, which base on the client-server architecture, are well suited for being spied, since almost all information about its users are gathered centrally at the providers' servers.

In contrast to the centralized server architecture, participants in a peer-to-peer network have basically equal rights. User-related information is distributed among the peers in a decentralized manner, without the need of a dedicated server. The peer-to-peer architecture is therefore suitable for sharing data or information dissemination without being stored on a central server. Nevertheless, as history shows, it is important for peer-to-peer solutions to be aware of network partitions, as they might occur

unexpectedly at any time, whether due to governments controlling the nation or because of natural catastrophes, which might disrupt parts of the Internet, as an earthquake did in Taiwan in December 2006 [9]. In case a network partitioning event occurs, participants of a disrupted peer-to-peer overlay should be capable of joining the common network again without administrative effort.

In this thesis we ascertain and evaluate criteria, which have to be focused on if ring-based peer-to-peer overlays are wished to be reunited again after network failures. Thus, we analyze already existing designs of overlay merging algorithms and introduce a novel merging algorithm, named Ring Reunion Algorithm, which is able to unify separated overlay networks autonomously and reliable.

1.2 Characteristics of Network Partitioning

Networks, especially the Internet which is used by most peer-to-peer applications, constitute no reliable infrastructure per se, even less reliable are the users of these services. Hence, out of different reasons, one or multiple nodes in an overlay, we take Chord as an example, may be separated from the other members. This partitioning can be divided into two cases.

- Single nodes are separated individually
- Whole groups of nodes are separated

1.2.1 Separation of Single Nodes

The first case occurs very often in many overlays and is therefore considered and handled in most structured overlays. This case includes all nodes, which fail suddenly, because they are separated shortly, or for a long period from the overlay, for example if one user leaves the overlay without warning, or if the connection between a node and the overlay is corrupted due to technical problems. These membership dynamics (churn) are handled in Chord by using a list of redundant contact nodes, which is denoted as successor list, so that in case of a failing node, Chord's function is not affected. If the previously failed node is capable of joining the overlay again, the Chord stabilize protocol will insert the node into the existing ring at the right place. All in all, the ability to treat churn in a proper manner is necessary to reconnect failing nodes with the overlay again. In the case of Chord, a single, global ring is maintained.

1.2.2 Separation of Whole Groups

The second case of partitioning is not considered in most overlays. It occurs when a whole group of nodes, which are not necessarily dependent from each other and do not correlate in a specific way on the overlay, but which are associated to a common group on network level, is separated from the existing overlay. This scenario could occur if all failing nodes are assigned to the same administrative domain, which departs from the common network, for example if a *geographical region* is disconnected from the Internet, because of a natural catastrophe like a heavy earthquake, or because an authoritative institution prohibits unrestricted access to the Internet. This case is more difficult to handle than the first case, inasmuch as a global ring might be impossible to achieve. To get a better impression of the explained scenario we have to discuss two more characteristics.

- Isolation

- Forming groups

Isolation

Whenever two or more geographical regions are separated from each other over a long period (*isolated*), it might happen that Chord's stabilize protocol will have replaced all outdated finger entries with newer ones, which are only present in the node's own region. Finger contacts that are part of the inaccessible region are deleted and overwritten completely. If the separated regions are reconnected again on network level, and both regions are capable of routing to each other, both regions will probably not converge to a common ring though. The reason for this behavior is obvious: not one node maintains any routing information about at least one contact node in the other region. Consequently, routing in a global overlay with both regions as participants is impossible. Both groups will remain independent from each other. In Figure 1.1 two connected regions are shown on the left. On the right, one region is isolated so that two rings are formed.

Forming Groups

When two regions are isolated from each other and no routing between them is possible, it may occur, that more than two rings will arise after a certain stabilize period. Although nodes of individual rings in the same region might be connected on network level, they could be incoherent in the overlay. Reason for this occurrence is the limited amount of redundant routing information which is maintained by the finger table of a single node. In a Chord ring almost all nodes are equally distributed, so that

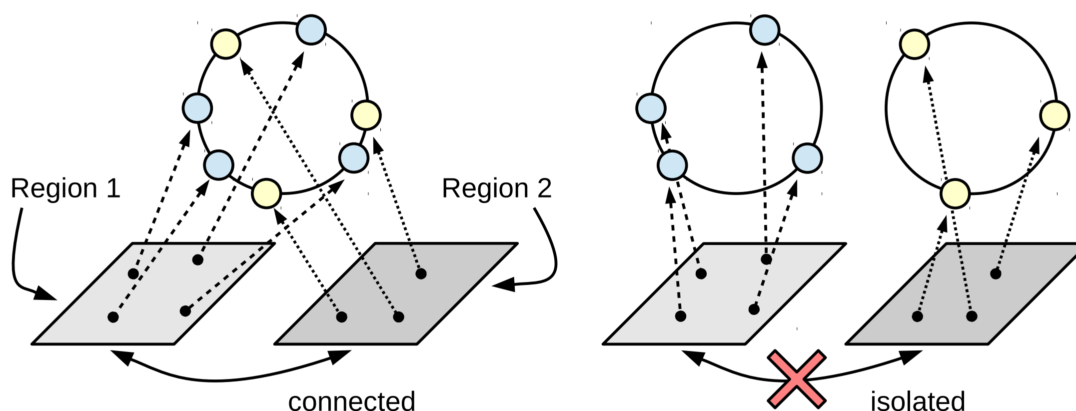


Figure 1.1: Ring is corrupted during isolation.

most participants of one specific region hold more routing information about nodes from the other region than about nodes from the own geographic region. Hence, during isolation only those nodes are formed to a group, which are represented in the same set of finger entries and successor or predecessor pointers. Figure 1.2 shows an example of a common ring which breaks apart. After one region (*A*,*C*,*D* etc.) became isolated, the remaining nodes might not be able to establish a common ring, as they only hold references to a small number of other nodes.

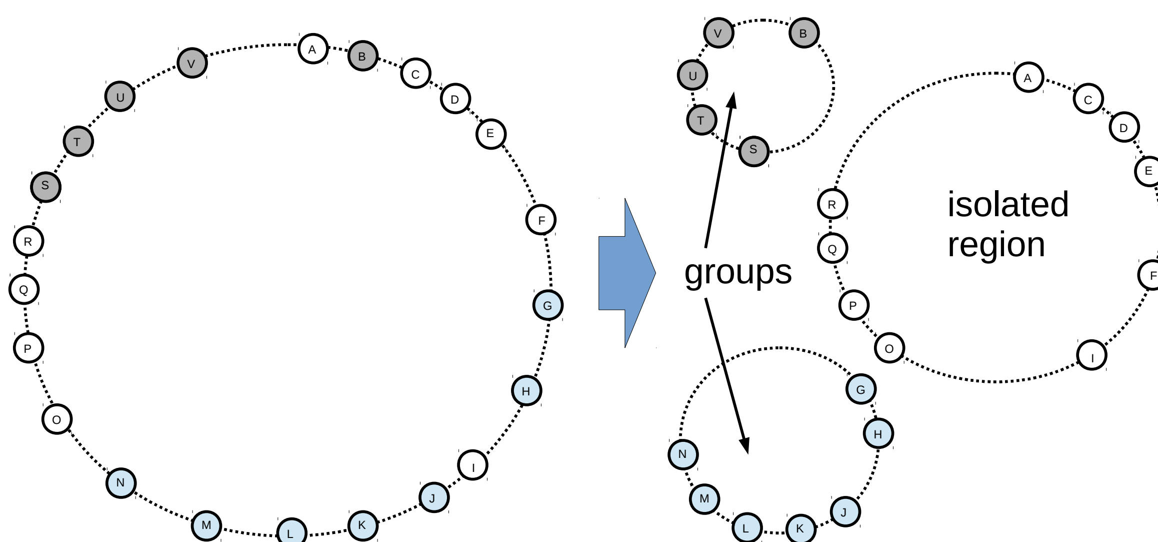


Figure 1.2: Groups are formed after one region has been isolated.

If an isolation lasts long enough and all finger entries are stabilized, several groups will remain, which form independent rings. Communication within these individual rings will be preserved and maintained. The functionality to include or even merge other rings is not given by default.

1.2.3 Requirements for Merging Algorithms

In order to preserve the principles of peer-to-peer systems, that is to distribute responsibility equally among all participants, merger algorithms should act locally on a specific node in a ring, without requiring global knowledge about the entire ring. In addition, any merger should operate independently, automatically and without administrative restrictions. Below, the most important criteria are summarized, which should be focused on, if multiple overlays are supposed to be merged.

Discovery of Contact Nodes

As mentioned above in Section 1.2.2, two groups do not have the ability to communicate with each other, if all routing entries are updated and only point into the node's own ring, even if routing on network level is possible after an isolation. Then it is necessary, that at least one node in one group (the *initiator*) gets in contact with a node from the other group (*contact node*), to start a merging algorithm.

Merging

If one merging algorithm is started by an initiator, the merger instance (token) should be passed from node to node, so that in the end, a global ring is created. Each node has to decide, which node is informed about the merging procedure next, how it handles information about other nodes, and how routing entries, successor pointers and predecessor pointers have to be treated and updated. The major goal of the merging algorithm is to modify and update routing information, that thereby a global, common ring arises. Each node only has to know its successor to preserve correct routing, whereas the adjustment of the additional routing table does not have to happen quickly. For the quality of a merging algorithm, it is essential, that each node forwards the merger token to a suitable successor, since this routing decision affects the function and performance of the entire merging algorithm.

Termination

Each merging algorithm should be terminated if a global ring is established. Instances of the current merger, which are supposed to be started or have been started earlier, should be terminated immediately, to avoid unnecessary messages and thereby bandwidth consumption. Further, each merging mechanism should be able to handle churn. On the one hand a merging algorithm should detect nodes,

which are separated from the global ring, and reconnect those nodes with the global ring, automatically. On the other hand any merging algorithm should be aware that separations of individual nodes could simply have been a result of churn, instead of isolated regions. In this case a quick termination is desired. Finally, it is useful to control the amount of running merger instances in order to regulate general costs and benefits. Each unnecessary merger instance should be stopped consequently.

Robustness

As mentioned previously, each merger should be able to handle departure and joining of single nodes properly. Moreover it is important to consider packet loss, so that a merger algorithm does not terminate early or multiple merger instances are executed uncontrollably. It might happen for example that messages, which are sent by the merger algorithm, get lost during the forwarding of the merger token, because the successor of one node fails. The respective node, which is currently responsible for merging, has to ensure that an other node is selected then to go on with the merger of the ring. Furthermore it has to be taken into account that the node fails, which is currently responsible for merging. All in all, message losses have to be considered and handled correspondingly.

Parallelization

Inasmuch as we focus on merging algorithms that act locally in a ring, without requiring knowledge about the global ring, these mergers can be improved by starting multiple merger instances simultaneously and in parallel. Doing this, each merger instance has to obey the above mentioned topics. In particular, it is very important that all merger instances terminate and do not hinder each other in their performance.

Message Complexity

To identify a good merging algorithm we also have to consider the message and bandwidth consumption of the respective algorithm. It is obvious, the less messages an algorithm produces, the less it affects the underlying network. In addition a merging algorithm should produce the less messages, the faster it merges separated overlays. Ideally, the message and bandwidth consumption should depend on the number of constructs which arise during the merger, instead of the number of participating nodes. One large ring should produce minimal overhead, whereas in multiple constructs the overhead is accepted to rise.

1.3 Outline

In this chapter we motivated our intend to merge different overlays with respect to partitioning events, and listed criteria which have to be met to do so. The rest of this thesis is structured as follows:

In Chapter 2 we introduce Chord as popular example for ring-based peer-to-peer overlays. We summarize one of the first works on the challenges of merging similar overlays in Section 2.2. As a next step, in Section 2.3, we analyze the studies which are presented in Section 2.2 and focus on existing Chord merging algorithms in the following sections. The Chord-Zip Algorithm [10] is summarized and investigated in Section 2.4 and the Ring Unification Algorithm [11] is described in detail in Section 2.5

In Chapter 3 we present a novel merging algorithm for ring-based overlays, named Ring Reunion Algorithm, which has been designed, implemented and extensively evaluated in this thesis. We explain the design of our Ring Reunion Algorithm in Section 3.1 and analyze it in comparison to the other approaches in Section 3.2.

In Chapter 4 the simulation environment, which has been used for our evaluation, is shortly explained. Furthermore, in Section 4.3, we nominate different metrics which are necessary to value the results of our simulation and to ascertain the quality of the different merging algorithms.

We evaluate our Ring Reunion Algorithm in comparison to the existing merger algorithms and the unmodified Chord stabilization protocol in Chapter 5. Therefore, we mainly focus on the ability of each algorithm to handle network partitioning events. Secondly, we try to identify mechanisms which increase the performance of a merger algorithm. A detailed description of the scenarios we simulated in this thesis, can be found in Section 5.1. The evaluation of the simulation results can be read in Section 5.2

Chapter 6 constitutes a conclusion of this thesis and gives a brief statement on further studies which are not considered in this work.

Chapter 2

Related Work

In this chapter we present existing studies on the challenges of merging different overlay networks. In Section 2.1.1 we focus on Chord as a representative work about ring-based overlays. One of the first studies about the merging of similar overlays is presented in Section 2.2. Furthermore, in Section 2.3 we analyze this study. Two existing approaches to unify separated Chord rings are summarized and analyzed in Section 2.4 and Section 2.5.

2.1 Background

Hash tables are capable of mapping keys to corresponding values, which are stored in a slot in an array. Independently of the number of entries in the array, the hash function calculates the position in the array at which an appropriate value to a given key is stored. In this way key-value pairs can be found in, inserted into, or removed from the underlying array in constant time. *Distributed hash tables* (*DHTs*) operate in a similar manner. Mainly, the difference to hash tables is that the underlying table, and therefore the responsibility for a given key, is distributed equally among multiple participants in a common network (see Figure 2.1). Thus, each member represents a part of the hash table, so that changes of the number of participants in the network hardly affect the performance of the distributed hash table. Even during the joining and leaving of individual members a distributed hash table is able to provide its service. In order to determine a responsible participant for a given key, routing information to close neighbors is stored by each member of the network. To increase routing performance, each participant maintains additional routing information to distant members in the network. Thus the search for responsible members is denoted as key-based routing. *DHTs* can mainly be characterized by their decentralized behavior: no server is needed and therefore a single point of failure is avoided, as well as by their robustness against membership failures, and lastly by their scalability in large-scale networks: each participant only stores a small fraction of information about the network. On top of this service one can imagine multiple types of applications (e.g. distributed databases, distributed

file systems, etc.). Popular implementations of distributed hash tables are Chord [12], Pastry [13], Kademlia [14], CAN [15], or P-Grid [16].

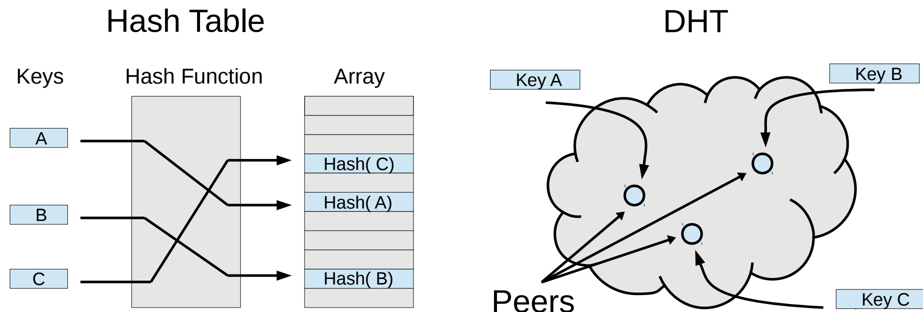


Figure 2.1: Hash Table and Distributed Hash Table.

2.1.1 Chord

Probably, Chord is the best known example for ring-based overlays, which implements a *distributed hash table* (*DHT*). Because of its simplicity, and its provable correctness and performance, Chord is inevitable for education and research on overlays. The Chord protocol supports only one operation: to find a node that is responsible for a given key. One could imagine an application on top of this protocol, which associates keys with data items, and stores those items in a distributed manner.

In Chord, each node, and each key is assigned to an m-bit identifier by a consistent hash function. All nodes are arranged on a circle, ordered by their identifier, and connected to the node with the next higher identifier. Below, the next node on the identifier circle is denoted as *successor*, the previous node on the circle is called *predecessor*. The last member is connected to the first, so that the circle (Chord ring) is closed. The most desirable characteristic of a consistent hash function is its ability to balance load with high probability. Therefore keys are distributed equally among participating nodes. Furthermore only $O(1/N)$ keys have to be moved to a different location if an Nth node joins, or leaves the Chord ring. Each key is assigned to the first node (*successor*) whose identifier succeeds the key's identifier, or is equal to it.

Key Lookup

Searching for a node, which is responsible for a given key, simply means to search for the key's successor. For a seeking node, the easiest method to find a key would be to initiate a query, that is passed around the Chord ring via the nodes' connections to its successors. To increase performance,

every node maintains an additional routing table with up to m entries (actually only $O(log\,N)$ entries are necessary within a circle of N participants), which is called the node's finger table. Its kth entry (k^{th} *finger*) holds information (identifier, IP address, and port) about the first node in the Chord ring that succeeds (the node's identifier + $2^{(k-1)}$) $mod\,2^m$, where $k \in [1,m]$. An advanced, and scalable method to find a responsible node is to pass a query to the largest *finger*, that precedes the target identifier. The node, whose successor is responsible for the given key identifier, returns the information about its successor via the previously traversed path. Each node is able to forward a query mostly halfway along the Chord ring, and therefore at least halves the distance between a node and the key identifier. Thus a query will be resolved in $O(log\,N)$ steps within a circle of N members.

Figure 2.2 shows the path of a lookup for key 58 which has been initialized by node 14. The lookup is forwarded via the closest preceding finger entries until the target node's predecessor (node 50) is reached. Finally, node 50 forwards the lookup to the target node (58).

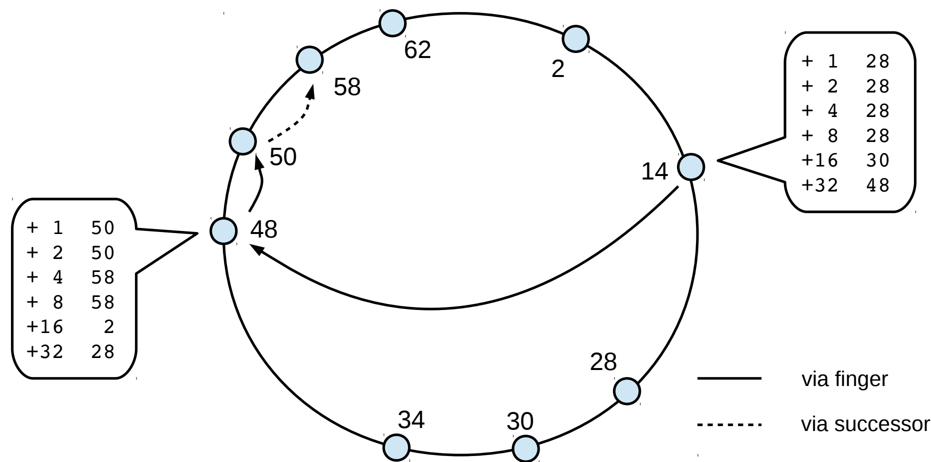


Figure 2.2: Chord Lookup: path of query for key 58, started at node 14.

Joining, Leaving, and Maintaining the Chord Ring

Chord needs to handle nodes, which join or leave the network suddenly. In a periodically changing environment it is necessary for all nodes, to hold valid pointers to their successors, since this is the only criterion to ensure correct lookups. Chord contains two mechanisms (*stabilize protocol*), which periodically update finger tables and successor pointers. Foremost are updated by frequently assigning each finger k to the current successor of (node's identifier + $2^{(k-1)}$) $mod\,2^m$. The latter are adjusted by asking one node's successor to return its predecessor. If this predecessor fits better as the node's successor, it is stored as the node's current successor. Nevertheless each participant is periodically informed by its preceding node about its assumption to be the node's predecessor. To join an existing

Chord ring, a node simply has to ask a familiar *contact node* to search for the successor of the joining node. After obtaining its successor, incorrect or missing finger entries and pointers are updated automatically by the methods mentioned above. In case a set of N nodes joins another set of N nodes, and all successor pointers are correct, lookups will still take $O(log N)$ time with high probability, before all finger entries and pointers are adjusted [12]. To increase further robustness, each Chord node maintains a list of its first r successors. When a node fails or leaves the Chord ring, suddenly, and without an announcement, lookups will be successful anyhow, because each node holds more than just one successor. Even if every node fails with a probability of fifty percent, the closest successor to any query key can be found with high probability in $O(log N)$ steps, assuming a list of length $r = O(log N)$ in a stable circle with N participants [12].

2.1.2 Chord Variants

Chord is one of the pioneering works on distributed hash tables and therefore has been optimized and extended in many ways. Below, examples of various chord extensions are listed to emphasize Chord's educational value.

B-Chord

In B-Chord [17] finger tables and lookups are modified to generalize Chord from one-sided lookups to two-sided lookups. In contrast to Chord, the finger table in B-Chord maintains successor and predecessor pointers. Lookup performance is improved by choosing the most fitting pointer for reaching a target node. This choice of a path with better locality leads to less routing costs and less routing latency. In Addition, B-Chord uses a combination of the physical distance between two nodes and their distance in the current overlay. By this means, B-Chord traverses less physical hops during lookups than Chord.

EpiChord

In EpiChord [18] a reactive routing state management strategy is used to decrease lookup costs. Finger tables in EpiChord are exchanged by caches, in which multiple successor and predecessor pointers are maintained. Routing information is extracted out of basic query replies and temporarily stored in the caches. In case that not enough queries are started to obtain suitable routing information EpiChord uses Chord's proactive approach to fill its routing table. Additionally, in EpiChord parallel queries are started to avoid lookup timeouts due to expired routing entries.

Re-Chord and Ca-Re-Chord

Re-Chord [19] is a distributed self-stabilizing protocol which extends Chord with the function to recover from any initial network state. Re-Chord extends a method called *local linearization* [20], which converts an arbitrary graph into a sorted list of nodes. The fingers of Chord are redefined as virtual nodes in Re-Chord. In addition, certain rules are used to stabilize the resulting Re-Chord network. If Re-Chord has reached a stable state, Chord can be seen as a sub-graph of Re-Chord. As Re-Chord is prone to churn during the stabilization process, it has been extended to a churn aware variant, introduced as Ca-Re-Chord [21].

S-Chord

S-Chord [22] is an extension to Chord, which enables bidirectional forwarding of messages within a Chord ring and routing in a symmetric manner. Finger tables in S-Chord are therefore divided into two parts. One part is responsible for traversing the Chord ring clockwise, the other part is responsible for traversing it anti-clockwise. On the contrary to Chord, in which finger entries are located at positive distances of powers of two, distances between nodes and its finger entries in S-Chord are of powers of four and guide into positive and negative direction. Hence, the size of the routing table in S-Chord remains the same as in Chord.

2.2 Tale of Two Networks

In [23] Datta et al. attempt to characterize challenges of merging two structured overlays, which base on the same protocol. The authors claim, the peer-to-peer research community had ignored the problem of merging two partitions of a structured overlay in the past. Reason for this omission would be the way most of the empirical information of peer-to-peer systems had been gathered. Experiments had been under a controlled setting with the intention to create only one overlay. None of those experiments had faced network partitioning problems, instead common bootstrap nodes had been used to ensure that separated overlays would not be created. Furthermore most of the knowledge about structured overlays has been derived from unstructured overlays, which can be merged easily since no peer has any specific responsibility. Peers that come in contact with each other establish neighborhood relationship and simply forward all messages they receive.

As the authors of [23] write, structured overlay networks have three characteristics in common. First, the key-space is distributed among peers, so that each peer is responsible for a specific domain. Secondly, a graph topology among the peers ensures efficient routing and connectivity, even under churn.

Thirdly, a routing algorithm enables the forwarding of messages, in order to serve a routing request. Therefore a structured overlay has to achieve two goals to be functionally correct. For one thing, it should be possible to reach the correct peers which are responsible for a given key (correctness of routing, achieved by routing protocol). For another thing, all peers should maintain all corresponding keys they are responsible for, and none other (correctness and completeness of keys-to-peers binding). The latter is achieved by proper synchronizing content among replica peers, and by moving keys to its corresponding peer, in case network changes occur.

A case study is done by the authors to compare ring-based overlays (Chord) with overlays, which use prefix-based routing and structural replication (P-Grid), as the authors describe the responsibility of multiple peers, which are mutual replicas then, for exactly the same key-space partition. The way Chord maintains its topology and handles churn provided good static resilience, but the merger of two ring-based overlays disrupted its operations [23]. The authors claim ring-based overlays could not operate correctly, until they were merged completely. On the contrary, structural replication in P-Grid provides robustness against churn and prevents the correctness of routing to be violated during network mergers. P-Grid enables access to all resources that have been accessible in a partition before, but access to all resources is only possible when all partitions are synchronized.

2.2.1 Case-Study Chord

Chord is described as one of the most extensively studied systems, that is mainly used in other overlays because of its well developed algorithms (for further explanation see Section 2.1.1). Besides, the authors of [23] define a ring network to be weakly stable if, for all nodes p, $predecessor(successor(p) = p$ is true, and strongly stable if in addition, there exists no peer s on the identifier space where $p < s < q$ and $successor(p) = q$. The ring invariant is violated when peers join or leave the network, but can easily be reestablished and become strongly stable within $O(N^2)$ rounds of stabilization if no churn occurs.

When two peers from different partitions (P_1 and P_2) would meet each other, they were not able to realize the difference of their origin, as the authors of [23] write. Thus both peers would replace their successor and predecessor pointers by the currently discovered pointers if they were suitable. This replacement would lead to a cascading effect, which caused all members to reconfigure its pointers. Any ring merger algorithm would work similarly to this effect, moreover, information about updated pointers had to be passed to the immediate neighbors, and churn had to be dealt with, too. Such a process would take $O(N_1 + N_2)$ steps, where N_1 and N_2 represent the number of peers in partitions P_1 and P_2.

Only functional correctness of routing was assured after the merger. Queries would be forwarded

to peers which were most likely responsible for a given key. To be able to locate objects that had previously been available in any individual network, the ring had to be reestablished and stabilized first. During the merger, the overlay's function would be completely destroyed. To gain access to all keys that had existed in any partition, it would be still necessary to move all keys to the peers, which had become responsible for it. If partitions P_1 and P_2 comprised N_1 and N_2 peers, then during a merger, the number of pointers to be corrected was approximately $N_1(1-e^{(-N_2/N_1)}) + N_2(1-e^{(-N_1/N_2)})$. As a consequence the minimum transfer of data from Partition P_1 to P_2 would be $N_1(1-e^{(-N_2/N_1)}) \cdot \alpha \cdot D_2/(N_1+N_2)$ where D_2 denotes the number of keys in P_2's key set $\mathscr{D}_2$ so that $|\mathscr{D}_2| = D2$, and α is the fraction of exclusive keys in both partitions ($|\mathscr{D}_1 \cap \mathscr{D}_2| = \alpha \cdot |\mathscr{D}_1 \cup \mathscr{D}_2|$).

2.2.2 Case-Study P-Grid

In P-Grid both, partitions and keys are represented as a set of m-bit identifiers, where m denotes the depth of a binary-tree (see Figure 2.3). Each peer corresponds to a leaf in a binary-tree, consequently each peer is associated with a path, which guides to a specific leaf. For search and routing functionality, for each level of the binary-tree, peers maintain references to other peers in complementary sub-trees, so that routing complexity is bounded by the tree's depth. Since references to peers are chosen randomly, different instances of a P-Grid network may exist for a fixed set of peers. As a result, routing has to be greedy in P-Grid. Furthermore multiple peers may be associated with exactly the same key-space partition to provide fault-tolerance. Those peers are mutual replicas, which execute an anti-entropy algorithm to synchronize and update their content.

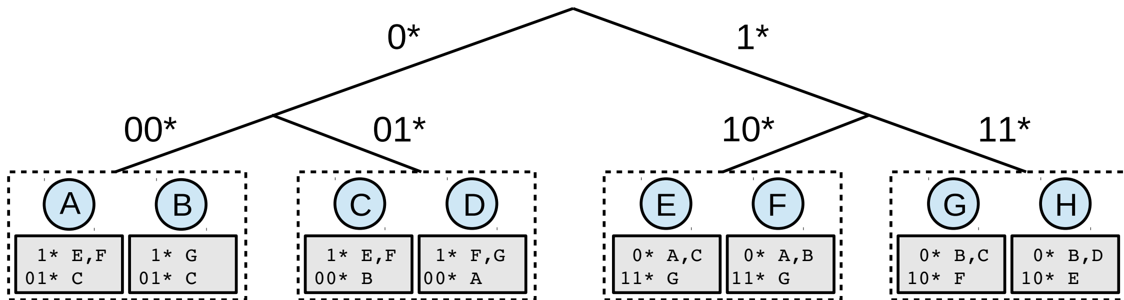


Figure 2.3: Example of a P-Grid instance.

Whenever two peers, with different paths, meet each other in P-Grid, they exchange information about other peers to improve routing efficiency. Peers which share the same path in the prefix tree execute an algorithm to combine their content and become mutual replicas. For those peers it is difficult to obtain knowledge about its replicas, since there is no proper structure among those peers. Thus the algorithm is rather probabilistic. Despite this fact, the synchronization algorithm has to be performed

on each replica, which is responsible for the same key space. In case one peers' path is prefix of the other peers' path, the peer with the shorter path can simply use the network joining algorithm to extend its path. In any case, when new peers join the overlay as structural replicas or existing peers, routing is not affected, and originally accessible keys (before the merger operation) will always be found (during the merger operation), since no routing information needs to be updated nor is affected by the joining process. All keys that previously have been available in other networks are accessible to all peers after the synchronization is finished. Adding reference to other peers is done in background, without interrupting the function of the overlay, too. Further on, no data has to be transferred among peers in case of changes in the key-space partition a certain peer is responsible for.

2.3 Analysis of P-Grid

The work of Datta et al. [23] is one of the first elaborations that introduce the idea of merging similar overlays and focuses on problems which have to be faced with the design of merging algorithms. The author's goal is to "explore the design space to better identify the features of overlay networks that can either facilitate or hinder merger of overlays - and hence get a better insight for (re-)designing such systems" [23]. To achieve this goal the authors compare Chord to P-Grid and describe the differences between those overlays. Nevertheless, they do not investigate any criteria which characterize the merging of ring-based overlays. Finally, their central statement about Chord is that "such a merger operation of ring topology based overlay will typically cause a complete interruption of the overlay's functioning" [sic] [23] and therefore lookups were not possible during a merger, even if they are executed in either of the previously stable rings. P-Grid, which by the way is developed by the same authors, was better suited to handle network mergers, as the authors of [23] write.

In Figure 2.4 the path of both nodes Y and Z is extended to 10* and 11*, respectively. Information about the new path can be obtained from nodes E,F,G or H. Other peers do not have to update their routing tables necessarily, but they might do so over time as a background process.

To defend Chord, one has to mention that multiple nodes in P-Grid are mutual replicas, so that it is no surprise that lookups to specific nodes are always possible, even during a merger. In fact the P-Grid merger is not a real merging algorithm since no routing information is changed during the merger. Only in case the size of the overlays which are to be merged differs, routing information is added to some nodes in the initial smaller overlay.

Another fact is that Chords goal is to distribute responsibilities for specific keys among all peers in the overlay. Against that, in P-Grid multiple nodes share the same responsibility for a specific key and no proper structure exists among the replicas. Thus two different problems might occur in P-Grid:

- No robustness in one instance: because of the randomly chosen routing information, one path might not be reachable if one node fails. If node C in instance N_1 fails (Figure 2.4), path 01* can only be reached from node H which is not present in any other routing table.

- Partitioning might occur without being detected: not all replicas might be reachable and accessible. In Figure 2.5 nodes A,D,F and G are connected to each other and nodes B,C,E and H form another instance. Thus two P-Grid instances are built independently. Node X is able to reach both instances, but no other node is aware of node X's existence.

Replica synchronization in P-Grid is probabilistic. Hence, the overlay is not well suited as a basis for further studies. Although the authors of [23] reveal a first idea which problems might occur during a network partitioning event and during the merger of an overlay, they do not give any advice, but rather draw a quick comparison between two different overlays.

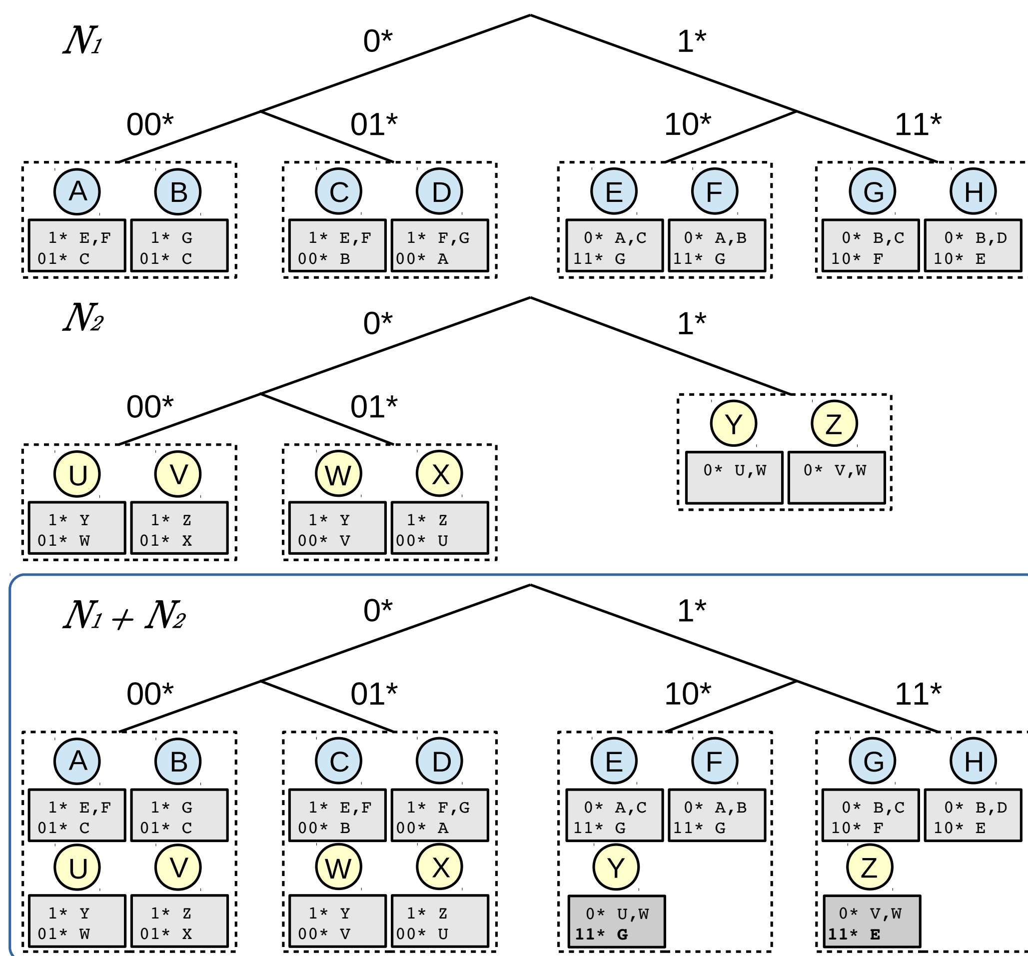


Figure 2.4: State of P-Grid instances after merger.

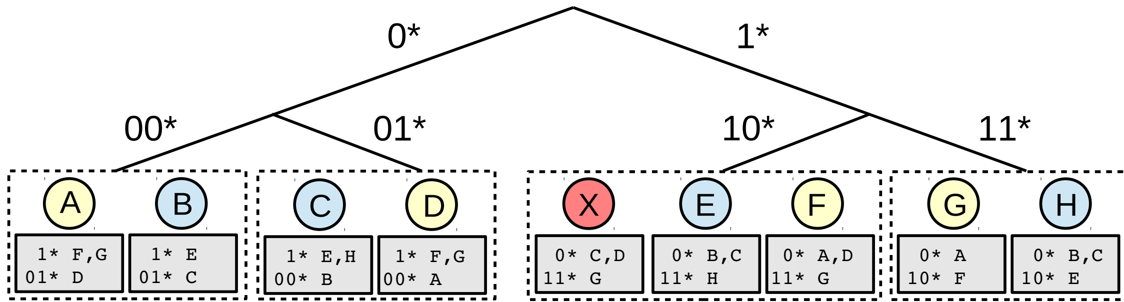


Figure 2.5: Example for partitioning in P-Grid instance.

2.4 Chord-Zip

In this section we present the preceding work about the Chord-Zip Algorithm [10]. In Section 2.4.1 we summarize the studies on the Chord-Zip Algorithm. Further, we present a more detailed description of the Chord-Zip Algorithm from Section 2.4.2 to Section 2.4.6.

Unfortunately, there are no code listings or more detailed explanations about the Chord-Zip Algorithm in [10]. Thus, the Chord-Zip Algorithm is reconstructed in this work by considering all available information, as far as it is possible. A notation in pseudo code of our interpretation of the Chord-Zip Algorithm can be seen in Listing 2.1. Although the Chord-Zip Algorithm is not implemented completely in this work, its reconstructed approach gives many insights on difficulties that have to be considered while designing a merger algorithm.

2.4.1 Summary of Chord-Zip

The authors of [10] suggest that enough information, required to merge two Chord rings, is already stored at any peer, which belongs to either of the rings, so that each node is able to rearrange its locality without global knowledge. In case an *initiator node* becomes aware of any other *contact node* in another ring, it will be able to start a query for its successor in the other ring (denoted as *alternative successor*). Thereafter, the initiator node updates its next neighbors (predecessor, successor) and asks the *alternative successor* for its predecessor, its successor list, its finger table, and for the data space partition it will be responsible for after the merger. Finally, the *initiator node* combines its successor list and its finger table with those, it obtained from the *alternative successor*, and passes its original routing information to the *alternative successor*. The merging process is repeated by every node, until it arrives the *initiator node* again, so that the merger is finished.

The execution time of the merger, which is composed of one Chord join operation and the merge signal, can be improved by determining different initiator instances that start the merger algorithm individually. Initiator nodes block bypassing signals in order to terminate the algorithm. For parallelization, the authors introduce two methods. First, nodes in both rings could start the merger algorithm concurrently. Secondly, some nodes could have the privilege to appoint further initiator nodes (for example finger entries).

In case the node with the merger token detects a broken connection to its (alternative) successor, it can choose the next node in the successor list for merging. If the node with the token fails, the initiator node can start a new merger instance after a specific timeout. Ways to handle keys, present in both rings, could be to select one key, or return both upon a request.

2.4.2 Discovery of Contact Nodes

The authors of the Chord-Zip Algorithm give no solution to this problem. They rather refer to external applications or neighbor selection algorithms which find proper candidates in the other ring by themselves. Using such an algorithm might not perform good as the number of contact nodes is limited to close neighbors only.

2.4.3 Merging

In Chord-Zip an initiator node starts the merging process. This initiator starts a simple Chord lookup (line 2 in Listing 2.1) to a contact node and receives its alternative successor in the other ring, which is to be merged. With the newly found alternative successor, the initiator can exchange routing information and forward the merger token to unify further nodes. Therefore each node sends its successor, successor list and finger table to its alternative successor (see lines 4 and 14 in Listing 2.1). Each receiver of such a Zip-Ping message answers with a Zip-Pong message which contains predecessor, predecessor list and finger table of the respective node in their original states. The merger token is forwarded by the initiator to its alternative successor, which has been obtained from the initial lookup. Other nodes, which are not an initiator, treat those successors as their alternative successors, which were send via Zip-Ping message to the respective node. By this means, the merger token will be forwarded through both rings, which are to be combined. The authors of [10] do not consider the case in which the identifier of the alternative successor follows the identifier of the own successor on the Chord ring, regrettably. Figure 2.6 shows node n which does not consider its current successor as next node although it is closer to node n than the alternative successor. As a result, n's successor is skipped and not properly merged. The chord stabilize protocol is needed then to connect n's successor afterwards.

Listing 2.1: Chord-Zip Algorithm.

```
1  receipt of STARTZIPPER(contact) from m at n
2     sendto contact : LOOKUP (n.id)
3     if altSucc is received from contact within time interval γ then
4        sendto altSucc : ZIPPING (n.succ, n.succList, n.fingerTable)
5        n.isInitiator := true
6     end if
7  end event

8  receipt of ZIPPING (m.succ, m.succList, m.fingerTable) from m at n
9     sendto m : ZIPPONG (n.pred, n.succList, n.fingerTable)
10    if n.isInitiator = true then
11       combine()
12       n.isInitiator := false
13    else
14       sendto m.succ : ZIPPING (n.succ, n.succList, n.fingerTable)
15    end if
16 end event

17 receipt of ZIPPONG (m.pred, m.succList, m.fingerTable) from m at n
18    if n.isInitiator = false then
19       combine()
20    end if
21 end event
```

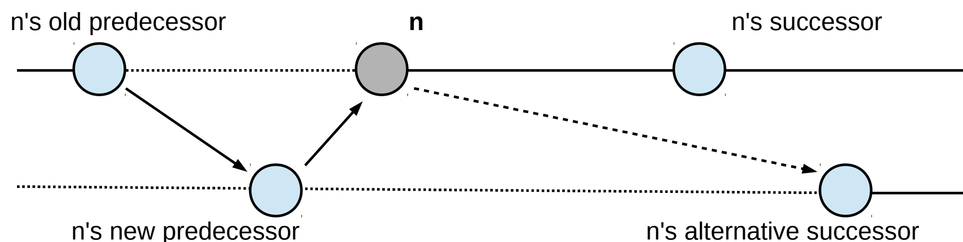


Figure 2.6: Node n skips its own successor in Chord-Zip.

Finally, each node obtains routing information from its alternative predecessor and from its alternative successor. This information can be compared to the node's own routing information and both can be combined to achieve best results (lines 11 and 19 in Listing 2.1). Doing this, all successor and predecessor pointer will be updated and every finger entry will contain the most fitting routing information. The initiator node has to wait for the Zip-Ping message of the last node, which executes the merger algorithm, before it is able to combine all routing information (see Figure 2.7).

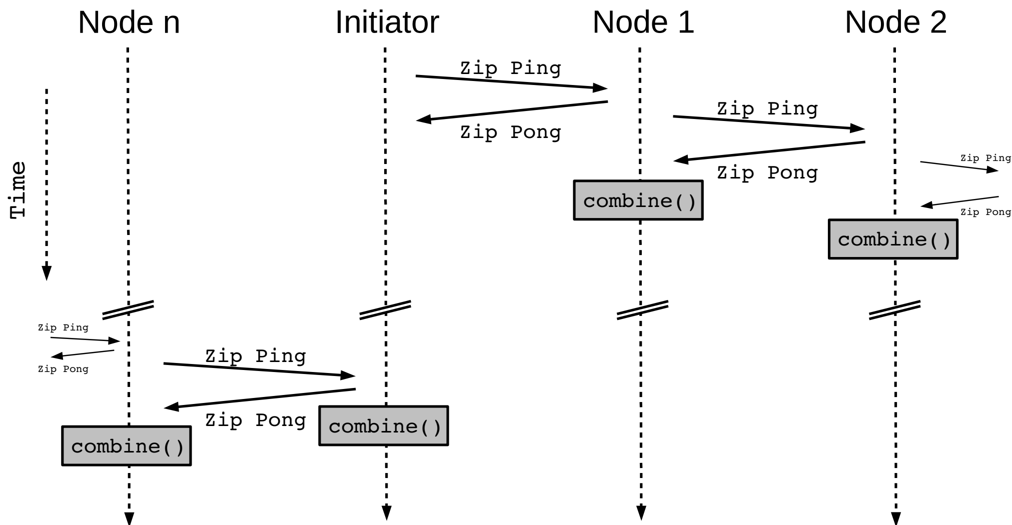


Figure 2.7: Messages in the Chord-Zip protocol.

2.4.4 Parallelization

The authors of the Chord-Zip Algorithm as well suggest to take advantage of the merger's locality by starting multiple Chord-Zip instances in parallel. Therefore multiple initiator nodes had to be determined simply, which in addition should block bypassing signals. As an alternative only one initiator could be defined and other nodes could have the privilege to ascertain supplementary initiator nodes, for example by using their finger entries. Precise explanations how the parallelization might be realized are not given by the authors in [10]

2.4.5 Termination

The authors of [10] suggest that initiator nodes should block bypassing merger tokens, so that if multiple instances of the Chord-Zip merger existed, each initiator could stop the merger token immediately and hinder it from being passed to nodes, which have been merged already. Unfortunately, the authors of [10] do not describe how long an initiator node should hold this state and block bypassing messages. If one initiator blocks messages for too long, it might additionally happen that it blocks merger instances, which should be forwarded by this node. Furthermore, the authors of [10] do not explain how one node, which is not an initiator, should stop the merger algorithm, if, for example, a node in the own ring should be merged inadvertently. The Chord-Zip Algorithm has been tested by its authors without the presence of packet loss or churn. They "assumed 50 msec message delays and 5 sec stabilization periods, both with 5% jitter" during their simulations [10]. As described below,

in Section 2.4.6, the Chord-Zip Algorithm uses message retransmission in case of message timeouts or discarded messages. If one node receives multiple messages (because of retransmission) from the same sender, all those duplicated messages are treated like independent messages, which have to be forwarded. By this incident, multiple Chord-Zip instances are duplicated and started, but not stopped directly. Moreover, the amount of executed merger instances is not controllable any more. One could imagine an extension with which nodes mark incoming messages with a time stamp and keep them, until a specific timer expires. Duplicated messages and thereby duplicated merger instances could be detected and stopped. If TCP is used for communication, it would be possible to consider sequence numbers as indicator for retransmission and duplicated messages. In addition, this extension had to take into account that multiple rings could be merged simultaneously. To compare Chord-Zip with the other merging algorithms anyway, we do not use packet loss in simulations that include Chord-Zip, as the above mentioned extension could be realized somehow with further expenditure.

2.4.6 Robustness

The authors of Chord-Zip give good advice how packet loss should be handled. If one node detects that its alternative successor is failing, before it has been able to pass over the merger token, the corresponding node can chose an optional alternative successor from the list of successors, which has been received via Zip-Ping message before, and forward the merger token to it. In case the node fails, which is currently holding the token, the authors suggest that the initiator node should start a new merger instance after a specific timeout. As a positive side effect an other problem is circumvented: the initiator only knows one alternative successor, since this node has not received a Zip-Ping message with an affiliated successor list at the beginning of the merger. Although Kis et al. [10] consider packet loss in their merging algorithm, they do not incorporate loss in their simulations. In fact, the Chord-Zip Algorithm is not able to handle duplicated messages properly, which leads to an uncontrolled number of merging instances. Hence, the Chord-Zip Algorithm might never terminate completely.

2.5 Ring Unification Algorithm

In this section the simple Ring Unification Algorithm and the gossip-based Ring Unification Algorithm [11] are summarized in Section 2.5.1. From Section 2.5.2 on, the Ring Unification Algorithm is examined with respect to the merging criteria listed in Section 1.2.3. The main difference of this algorithm to the other approaches is its ability to operate bidirectionally by starting modified lookups rather than forwarding a merger token via the ring.

2.5.1 Summary of the Ring Unification Algorithm

The authors of [11] regret that the aspect of network mergers had been ignored in the context of structured overlay networks. Although large-scale peer-to-peer systems were highly related to the problem of network partitions and mergers, those systems had just been studied under frequent joins and failures of nodes (*churn*). Furthermore it was widely believed that ring-based structured overlays were intrinsically ill-suited for network mergers.

Therefore the authors of [11] focus on the problem of dealing with partition mergers at the routing level and present an algorithm for merging any number of similar structured, unidirectional ring-based overlays. Their presented algorithm uses a parameter to adjust the trade-off between message complexity and time complexity, called *fanout* parameter. Whenever a peer detects that another peer in its routing table is not reachable any more, the routing information of the failed node is moved into a *passive list*, which is maintained by each node individually. In order to unify a partitioned overlay again, each node periodically tries to get in contact with nodes from its *passive list*. If one node is detected to be reachable again, a ring merging algorithm is started on both nodes. In case two different network partitions had never been part of a common network, and thus no partition is registered at one of the other partition's *passive lists*, a system administrator can insert an active node into the *passive list* of any node in the other ring to start a network merger manually. To avoid false-positives, every node that joins the network generates a globally unique random nonce. If one node is detected by the algorithm to be alive again, the node's current nonce is compared with that in the *passive list* to distinguish previously failed nodes from new nodes, which coincidentally have the same overlay and network address.

Simple Ring Unification Algorithm

The simple Ring Unification Algorithm uses a queue, which is periodically checked by every node and contains any alive nodes detected in the passive list. The Ring Unification Algorithm is then started on the newly found node as well as on the node which detected the other node. Both nodes are advised to start a lookup to the other node in their own ring. Similar to lookups in Chord, the lookup request is passed through the ring via the closest preceding fingers of the target identifier. The node which is responsible for the identifier itself or whose successor is responsible for the identifier, instructs the node from the other ring to consider the responsible node and its predecessor or the responsible node and its successor, as its new neighbors. The target node then starts lookups to its possible neighbors and updates its predecessor and successor afterwards. As a result the merger proceeds in both directions, clockwise and anti-clockwise.

Gossip-based Ring Unification Algorithm

In addition to the simple Ring Unification Algorithm the gossip-based Ring Unification Algorithm starts multiple instances of the merger algorithm at random nodes with uniform distribution. The enhancement should increase the algorithm's performance during churn and other pathological scenarios that would immediately terminate the simple Ring Unification Algorithm. The fanout parameter is decreased every time a random node is picked, to ensure that a constant number of merger instances is created and to avoid that too many messages are produced. After evaluating the algorithm with different parameters, the authors suggest a fanout value around 3-4 as a good trade-off between message and time complexity.

2.5.2 Discovery of Contact Nodes

In [11], the simple and gossip-based Ring Unification Algorithm use a passive list, which detects nodes that are suitable contact nodes. Therefore, all nodes, that are suddenly detected to be unreachable, are inserted to the passive list, which is maintained by every node. Each node periodically pings contact nodes in its passive list. If contact nodes are detected to be alive again, they will be removed from the passive list and inserted into a queue, out of which merger algorithms are started. One disadvantage of such a passive list is, that only those nodes can be merged, which had been known previously. Changes of the routing information in each individual partition might make the content of the passive list useless, because information about previous contact nodes might be outdated and some nodes may not exist any longer. In that case, a system administrator could insert a contact node to a specific node manually, and thereby start a merging algorithm.

2.5.3 Merging

The Ring Unification Algorithm (Listings 2.2 and 2.3 from [11]), which in its basic form consists of two parts, completely differs from the Chord-Zip Algorithm. When using the Ring Unification Algorithm, the initiator commands the contact node to start a merger, as well. The contact node uses the initiator node as its own contact node, so that both nodes try to merge each other respectively.

The merging process is, by its nature, started in both rings, so that two instances are executed in parallel. The method *MLookup* carries the merger token closer to the contact node via the closest preceding finger method, which is used in Chord, too. If then the node, which is to be merged, is located between the node, which holds the merger token, and its predecessor or its successor, the method *TryMerge* is called. In *TryMerge*, a node examines if the contact node fits better as predecessor

Listing 2.2: The Simple Ring Unification Algorithm.

```
1  every γ time units and detqueue ≠ ∅ at p
2      q := detqueue.dequeue()
3      sendto p : MLOOKUP (q)
4      sendto q : MLOOKUP (p)
5  end event

6  receipt of MLOOKUP (id) from m at n
7      if id ≠ n and id ≠ succ then
8          if id ∈ (n, succ) then
9              sendto id : TRYMERGE (n, succ)
10         else if id ∈ (pred, n) then
11             sendto id : TRYMERGE (pred, n)
12         else
13             sendto closestprecedingnode(id) :
                       MLOOKUP (id)
14         end if
15     end if
16 end event

17 receipt of TRYMERGE (cpred, csucc) from m at n
18     sendto n : MLOOKUP (csucc)
19     if csucc ∈ (n, succ) then
20         succ := csucc
21     end if
22     sendto n : MLOOKUP (cpred)
23     if cpred ∈ (pred, n) then
24         pred := cpred
25     end if
26 end event
```

Listing 2.3: Gossip-based Ring Unification Algorithm.

```
1  every γ time units and detqueue ≠ ∅ at p
2      ⟨q, f⟩ := detqueue.dequeue()
3      sendto p : MLOOKUP (q, f)
4      sendto q : MLOOKUP (p, f)
5  end event

6  receipt of MLOOKUP (id, f) from m at n
7      if id ≠ n and id ≠ succ then
8          if f > 1 then
9              f := f − 1
10             r := randomnodeinRT()
11             at r : detqueue.enqueue(⟨id, f⟩)
12         end if
13         if id ∈ (n, succ) then
14             sendto id : TRYMERGE (n, succ)
15         else if id ∈ (pred, n) then
16             sendto id : TRYMERGE (pred, n)
17         else
18             sendto closestprecedingnode(id) :
                       MLOOKUP (id, f)
19       end if
20     end if
21 end event

22 receipt of TRYMERGE (cpred, csucc) from m at n
23     sendto n : MLOOKUP (csucc, F)
24     if csucc ∈ (n, succ) then
25         succ := csucc
26     end if
27     sendto n : MLOOKUP (cpred, F)
28     if cpred ∈ (pred, n) then
29         pred := cpred
30     end if
31 end event
```

or successor than the current pointer. For better comparability, lines 18 and 22 in Listing 2.2 and lines 23 and 27 in Listing 2.3 will be replaced by a direct method call, so that no delays occur when sending messages via the local loop-back interface. In fact, this delay could hinder the algorithm to perform properly, as the successor pointer would be updated (line 25 in Listing 2.3), before *MLookup* is executed, and therefore the merger algorithm, which operates in clockwise direction, would be terminated early. Basically, the passive list or the choice of any random contact can be used to find a contact node in the other ring, since the merging mechanism is separated from the technique to find proper contact nodes.

2.5.4 Parallelization

The gossip-based Ring Unification Algorithm takes advantage of the possibility to start multiple merger instances as well. To achieve that all initiator nodes are distributed equally, the nomination of new initiator nodes is integrated in *MLookup* (lines 8-11 in Listing 2.3). With each step in *MLookup*, a new initiator node is selected randomly out of the own routing table, and a command to start a merger instance is sent to it. The number of additional instances is limited by the fanout parameter (using 1 as fanout value is equally to use the simple Ring Unification Algorithm). Furthermore the usage of multiple instances increases the algorithm's robustness during churn and the following pathological scenario is circumvented: if successor and predecessor pointers of all nodes point into the own ring, but the additional pointers refer to the other ring, an *MLookup* will leave the own ring and terminate the merger immediately.

2.5.5 Termination

In both Ring Unification Algorithms the merger algorithm is kept running by frequently starting lookups to neighboring candidate successors or predecessors (*TryMerge* in Listing 2.2). Only if a node is advised by another node to start a lookup to itself or to its successor, the merging algorithm is stopped (see *MLookup* in Listing 2.2). Finally, if multiple rings are successfully merged, the respective node which holds the merger token, sends a *MLookup* with its successor as target to itself, so that its successor becomes the candidate successor in *TryMerge* and the merger instance will be terminated thereupon. As mentioned before, the original algorithm would result in prematurely updated successor pointers, so that the merger would terminate early (consider lines 18-21 in Listing 2.2). In contrast to the authors of the Chord-Zip Algorithm, the authors of the Ring Unification Algorithm do not discuss packet loss and retransmission in their paper. But since the introduced $detqueue$ (line 2 in Listing 2.2) periodically tries to merge available nodes, a concept for retransmissions in case of delayed or discarded messages is not necessary.

2.5.6 Robustness

Although the authors of [11] do not consider packet loss in their study, the Ring Unification Algorithm is able to cope with loss, since the passive list detects communication failures periodically. Failing nodes and messages that get lost are therefore spotted and the passive list is able to react thereupon. Alive nodes can be merged then, as soon as they are given to the queue ($detqueue$) of the Ring Unification Algorithm. Even under churn the algorithm is able to behave as predicted, without the need to distinguish churn from isolation scenarios. Like the Ring Reunion Algorithm, the Ring Unification

Algorithm terminates early if nodes in the own ring are suggested as contact nodes for the merger algorithm. In principle, the Ring Unification Algorithm should therefore be able to merge multiple rings simultaneously, too.

Chapter Conclusion

In this chapter we have presented related work and background information on the topic of overlay mergers. We summarize Chord in Section 2.1.1 as it is the best studied ring-based overlay we know. Furthermore, Chord has been object of previous investigations as described in Section 2.2. The authors of [23] claim that routing in ring-based overlays, especially in Chord, would not be possible during a merger. Furthermore, we have analyzed an overlay, the authors prefer, in Section 2.3 and have shown that this overlay can not be compared to Chord directly.

Two existing algorithms have been introduced and further examined in Section 2.4 and in Section 2.5. Doing this, we have focused on certain criteria which have to be paid attention to if separated overlays are supposed to be unified. It has to be clear how reachable nodes are discovered again, how the information of an ongoing merger operation is disseminated, and finally it is necessary that a merging algorithm terminates properly. Since the presented algorithms act locally in a Chord ring, it is possible to parallelize the mergers.

Next, in Chapter 3, we present the Ring Reunion Algorithm, a novel merger algorithm for ring-based overlays which has been developed as part of this thesis. In Section 3.1 we examine the design of our algorithm with respect to necessary merging criteria and later, in Section 3.2, we compare the Ring Reunion Algorithm to the mergers which have been presented in this chapter.

Chapter 3

Ring Reunion Algorithm

In the previous chapter the most important studies on the topic *'merging of networks'* have been introduced. Primarily the work of Datta et al. [23] gives a first insight on difficulties that have to be faced if two similar overlays, especially ring-based overlays, are merged. Kis et al. [10] and Shafaat et al. [11] extend these insights by developing different merging techniques to show that it is possible to merge ring-based overlays. With the knowledge about existing merging algorithms, which are described in detail in Section 2.4 and in Section 2.5, a new merging mechanism has been designed, implemented and evaluated in this work. In this chapter we introduce our newly developed Ring Reunion Algorithm. A detailed description about the design of our approach is given in Section 3.1. In addition, Listing 3.2 shows an elaborate implementation of the Ring Reunion Algorithm in pseudo code. This work is a first approach to analyze the existing solutions and to compare those approaches to the Ring Reunion Algorithm. Thus, Section 3.2 analyzes the performance of the different algorithms.

The reason why Chord has been chosen as an example for ring-based overlays in this work is that the above mentioned studies base on Chord and mostly all knowledge about overlay mergers is derived from those. Furthermore Chord is one of the most famous overlays, well studied and has a high educational value. In contrast to Chord, the idea of merging similar overlays is much younger. This work should be seen as a contribution to this topic, to gain better insights on the merging of ring-based overlays, which might be transferable to other structured overlays, like unstructured overlays had enriched the knowledge about structured overlays a few years ago. Unfortunately, most overlay designs make no or only little suggestions about how network partitions should be handled or might be merged. In general, the advantage of ring-based overlays is that they are very tolerant against failures and robust against membership dynamics (*churn*). Chord for example achieves its robustness through its *stabilize protocol*, which by itself reconnects the ring whenever individual nodes join or leave the overlay voluntarily or involuntarily. A simple approach to merge two similar overlays could be to send a signal through one ring which orders the corresponding nodes to leave this ring one after another and to join the second ring in turn. Chords stabilize protocol would then supervise the correct integration of all joining nodes.

3.1 Design of the Ring Reunion Algorithm

In this section we present the design of the Ring Reunion Algorithm which automatically detects network partitions and initializes merging operations thereupon. Below, in Section 3.1.1, we point out how the algorithm finds new contact nodes. Section 3.1.2 explains how our algorithm merges different network partitions. Furthermore, we focus on its ability to reduce traffic overhead (Section 3.1.3), we describe its ability to start parallelized instances autonomously (Section 3.1.4) and we point out how the algorithm terminates, in order to limit costly resources (Section 3.1.5).

3.1.1 Discovery of Contact Nodes

The idea of the passive list is picked up in this work and tested for all introduced merging algorithms. The authors of the Ring Unification Algorithms do not specify a size for the passive list or the queue in which detected alive nodes are kept (line 2 in Listings 2.2 and 2.3). Therefore we will test the passive list with unlimited size. Another method which will be tested in this work, is a list of random contact nodes, which every node could obtain from a bootstrap node during the joining of an overlay. Periodically, each node tries to get in contact with a random contact out of this *active contact list*, via ping pong messages. If one node is discovered to be alive and reachable, it is considered to be merged by the initiator. Doing this, it does not matter if the contact node lies in an other geographical region or in the same region as the initiator. Actually, during the isolation of two regions, no contact nodes in the other region can be merged, but as we saw earlier in Section 1.2, groups might be formed inside of a region. If then one node randomly chooses a contact node from the own region, but from another group, at least both groups are merged, with the goal that a global ring is created in this region (see Figure 3.1). The problem with the passive list is that two groups which are located in the same isolated region might not be merged, since those groups did not know about the other group before, which is furthermore the reason both groups have been formed independently from each other.

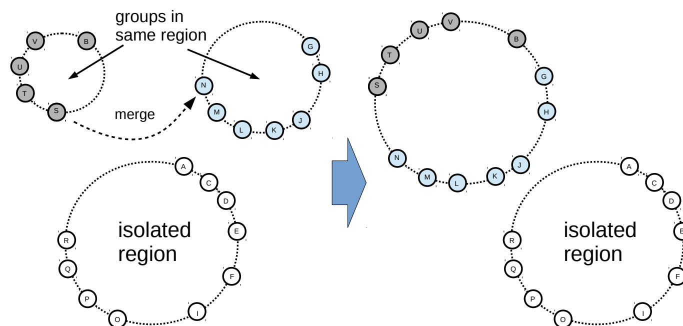


Figure 3.1: Example: groups in the same region can be merged during isolation.

3.1.2 Merging

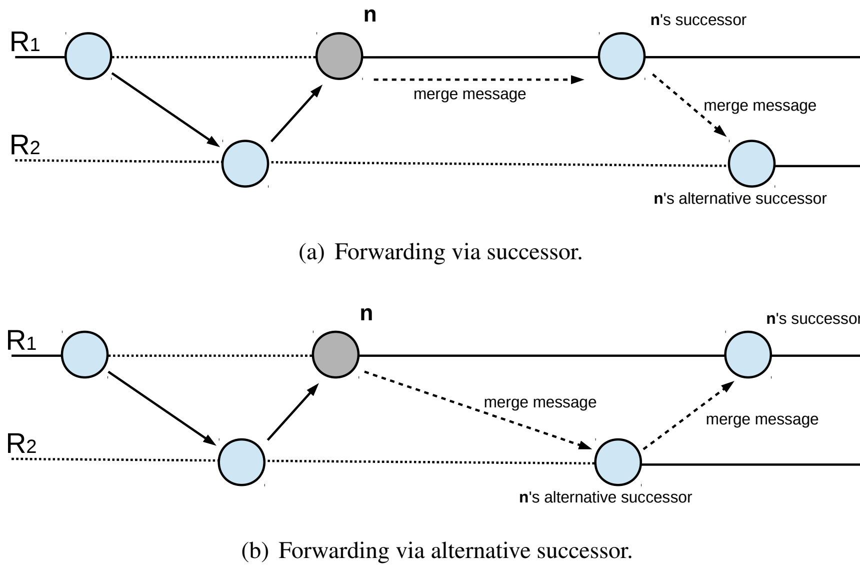
In this work the basic Chord-Zip merging mechanism is adopted, changed and extended. Likewise to [10], the Ring Reunion Algorithm starts a lookup to a newly found contact node, in order to obtain an alternative successor. For this purpose, the passive list or the *active contact list* can be used, as it can be with the Ring Unification Algorithm, since within the Ring Reunion Algorithm, merging mechanisms are separated from searching techniques, too. If then a contact node is found, the initiator starts the merging algorithm by sending its successor to its alternative successor (if $altSucc \in (n, succ)$, line 11), or by sending the alternative successor to its successor (if $altSucc \notin (n, succ)$, line 14 in Listing 3.1). In Figure 3.2 node *n* decides to send the *merge* message to the nearest succeeding node, which might be *n*'s own successor (3.2(a)) or its alternative successor (3.2(b)).

Listing 3.1: Ring Reunion Algorithm.

```
1   receipt of STARTMERGER(contact) from m at n
2      sendto contact : LOOKUP (n.id)
3      if altSucc is received from contact within time interval γ then
4         MERGE (altSucc)
5      end if
6   end event

7   receipt of MERGE (altSucc) from m at n
8      pred := m
9      if n ≠ altSucc then
10        if altSucc ∈ (n,succ) then
11           sendto altSucc : MERGE (succ)
12           succ := altSucc
13        else
14           sendto succ : MERGE (altSucc)
15        end if
16     end if
17  end event
```

The receiver of a *merge* message considers the contact information, which has been sent within the message, as its own alternative successor. The sender of the message is treated like the receiver's alternative predecessor and used as new predecessor, if it fits better than the current predecessor. By this way the correct order of the global ring is kept, and the merger token is passed through both rings, until they are merged. In contrast to Chord-Zip, the Ring Reunion Algorithm does not combine finger entries during a merger process, in order to prevent current routing behavior from being affected, as can be read in Section 3.2.



(a) Forwarding via successor.

(b) Forwarding via alternative successor.

Figure 3.2: Ring Reunion Algorithm selects suitable alternative successor.

3.1.3 Probability

We extend our algorithm by introducing a probability for starting a merger instance, in order to limit the number of merging processes and bandwidth consumption. Whenever an alive node is found by one nodes *active contact list*, a random value γ is picked. If one node is member of an overlay construct with size $size_c$, it starts a merger instance only if γ is below or equals $\alpha/size_c$, with parameter α, for which suitable values are specified in our evaluation in Chapter 5. If, for example, the probability is given by $1/size_c$, only one node per ring starts a merger instance on average. To estimate $size_c$, i.e. the size of the current network, we follow an approach similar to [24]. We calculate the average responsibility range, $resp_p$, of a node p by considering the identifier ranges of its successors and finger entries. The estimated number of nodes in the overlay construct is then given by $size_c = \frac{2^{160}}{resp_p}$. As a result every construct initiates α merger instances in a specific interval, instead of starting one instance per node. As shown in the evaluation, the merger performs well with this concept. In our comparison we use this method also in combination with the Ring Unification Algorithm, to examine its performance.

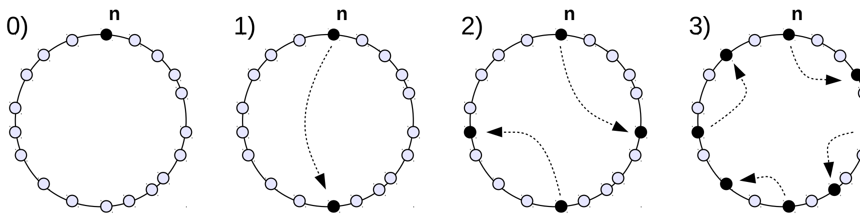


Figure 3.3: Affected nodes, in case the distribution algorithm is used.

3.1.4 Parallelization

To improve the Ring Reunion Algorithm, a further algorithm has been designed to start multiple merger instances simultaneously and independently (Listing 3.2). The *distribute* method is started every time a merger instance is started (line 2). As a result, the initiator node informs its furthermost finger contacts to initiate new merger instances. Thereupon those finger contacts inform their second furthermost finger contacts respectively et cetera, that thereby the information to start new instances is equally distributed among all nodes in the initiator node's ring (lines 7-10). Concurrently, a distributed counter is decreased with each message, so that exactly $2^{maxInstances} - 1$ additional merger instances are initialized, where *maxInstances* is a fixed parameter (see line 7). Hereafter all nodes that have received a *distribute* message start the *merge* method (line 11) and begin to merge a given ring (lines 14-24). Figure 3.3 demonstrates that with each step i, 2^i nodes are asked to start a merger instance so that, for example, after 3 steps $2^3 = 8$ instances are started.

Listing 3.2: Ring Reunion Algorithm with parallelization.

```
 1  receipt of STARTMERGER(contact) from m at n
 2     DISTRIBUTE (0, contact)
 3  end event

 4  receipt of DISTRIBUTE (cnt, contact) from m at n
 5     sendto contact : LOOKUP (n.id)
 6     if altSucc is received from contact within time interval γ then
 7        while (cnt < maxInstances) do
 8           sendto fingerEntry(lastEntryPos − cnt) : DISTRIBUTE (cnt + 1, contact)
 9           cnt := cnt + 1
10        end while
11        MERGE (altSucc)
12     end if
13  end event

14  receipt of MERGE (altSucc) from m at n
15     pred := m
16     if n ≠ altSucc then
17        if altSucc ∈ (n, succ) then
18           sendto altSucc : MERGE (succ)
19           succ := altSucc
20        else
21           sendto succ : MERGE (altSucc)
22        end if
23     end if
24  end event
```

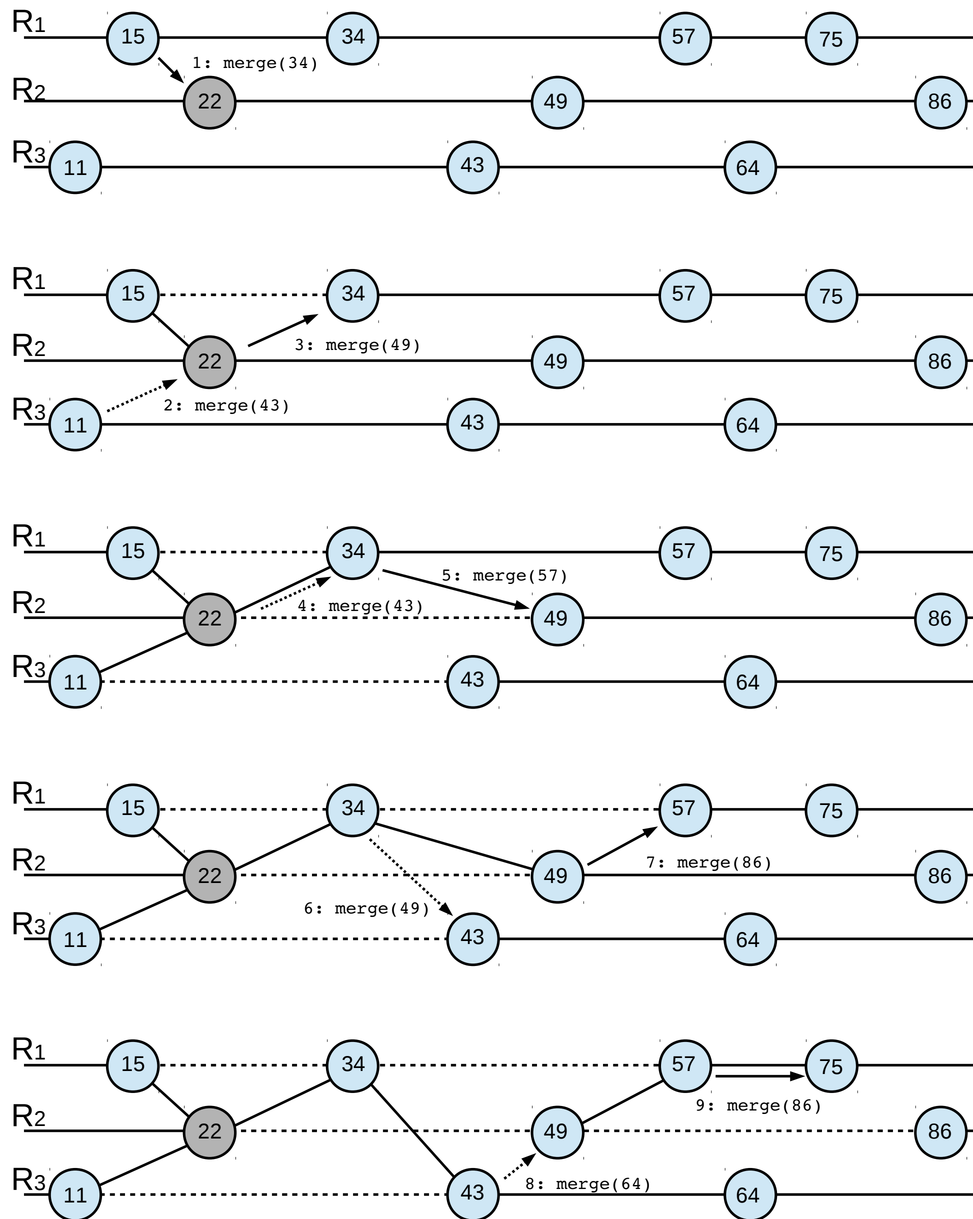


Figure 3.4: Verification of Ring Reunion Algorithm (1): two nodes try to merge ring at the same time.

In addition to the parallelization, the Ring Reunion Algorithm is capable of merging multiple rings simultaneously. Figures 3.4 and 3.5 show a scenario in which two nodes try to merge one node (22) at the same time. Since one *merge* message will always be received first, the second Ring Reunion instance will always merge a ring that has been merged previously. In this example node 11's *merge* message is received by node 22 shortly after node 15's *merge* message has arrived at node 22. Arrows describe forwarded *merge* messages. Solid lines denote the successor pointers from one node to its successor. When both merger instances are terminated, a single ring is created.

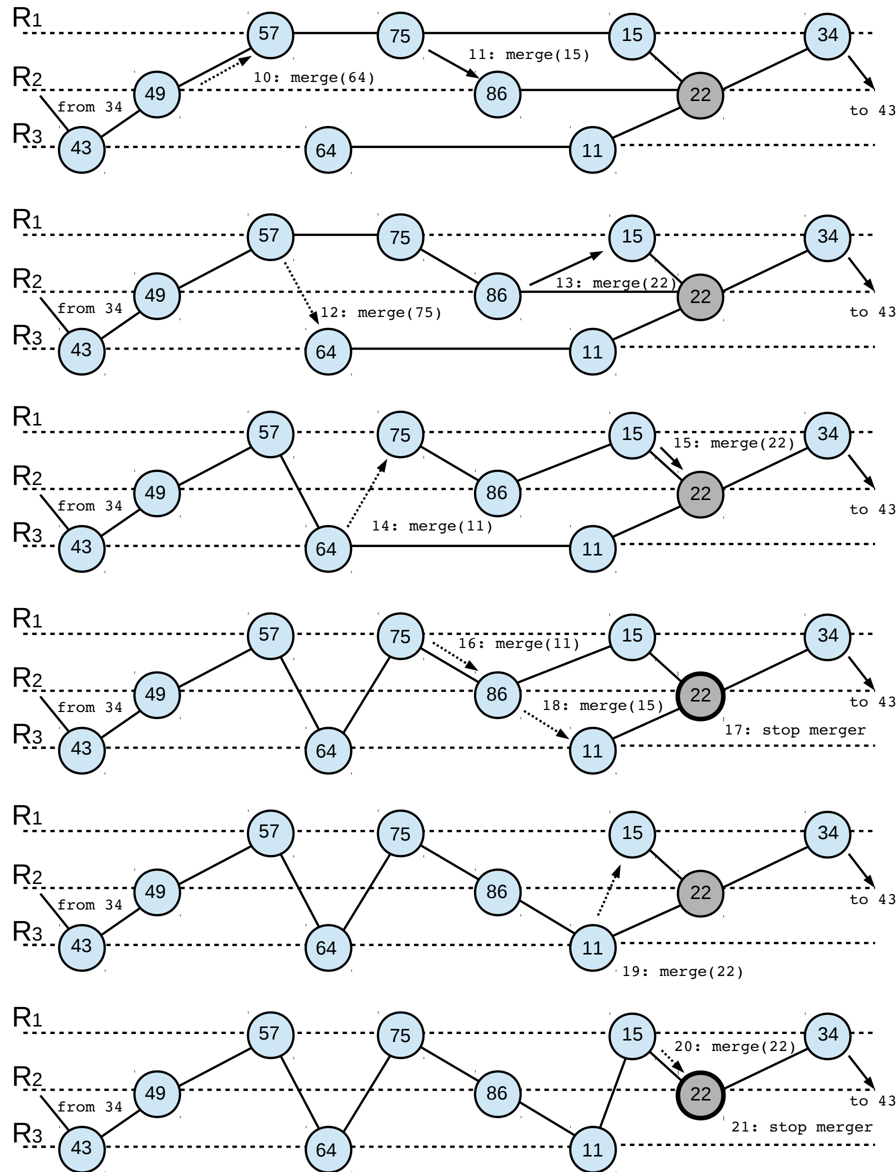


Figure 3.5: Verification of Ring Reunion Algorithm (2): two nodes merge ring at the same time.

3.1.5 Termination

The Ring Reunion Algorithm terminates if the received alternative successor is equally to the node which holds the merger token (line 16 in Listing 3.2). That is, the node has been commanded to merge itself. if then one node obtains a contact node from the own ring, a lookup is started in this ring to find the successor node of the node's id. Consequently the node receives itself as the successor node for its id, whereupon the merger instance is terminated. Because of this behavior, the Ring Reunion Algorithm recognizes if a given contact node is already merged so that it is able to stop immediately.

An example is given in Figure 3.6: node *n* came in contact with a node from its own ring and asks it for its alternative successor (1). The contact node starts a Chord lookup to find *n*'s alternative successor (2-3). The alternative successor is then given to *n* (4). Node *n* tries to merge the given node and stops immediately, as it is equal to the given alternative successor (5-6).

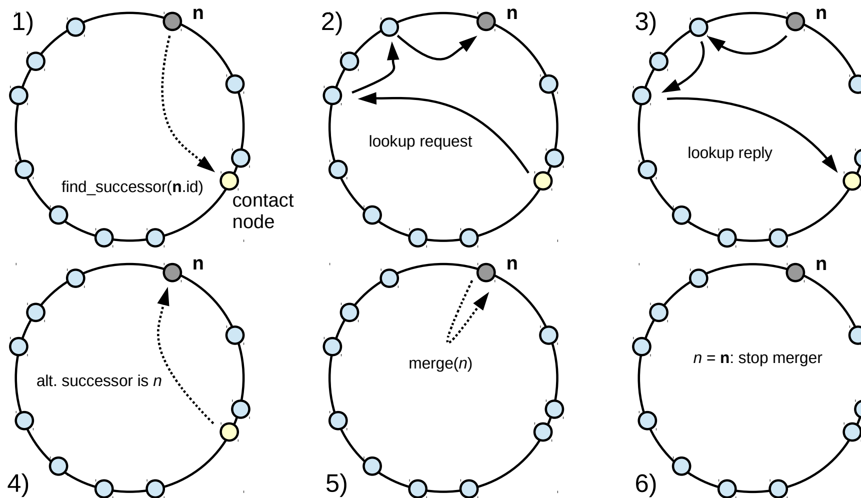


Figure 3.6: Example: one node tries to merge own ring.

3.1.6 Robustness

The Ring Reunion Algorithm has been designed in the way that it is capable of using acknowledgements optionally, if messages are forwarded. In case a merger message or a responding acknowledgement gets lost, a node can react quickly and retransmit the last message. If then one message is, because of timeout and retransmission, received by a node for the second time, the second merger instance terminates quickly, as the first merger instance (received via first message) already has merged the ring locally. Nevertheless, the robustness of the Ring Reunion Algorithm does not depend on acknowledgements. Like in the Ring Unification Algorithm, the passive list can be used here to start new, independent Ring Reunion instances, regardless of whether messages get lost and nodes fail or not. As an alternative to the passive list, nodes have the option to merge random nodes from the *active contact list*, which are found to be alive by pinging them. This method contributes to increase the merger's robustness, as different nodes are merged periodically. If one instance is terminated because of packet loss or churn, this method ensures that further merger instances are initialized, as soon as a suitable contact node is found. The *active contact list* is obtained by every node during the join process from the bootstrap manager. Additionally this list could be updated and filled with contact nodes which have been found via IP-scans.

3.2 Performance Analysis

All presented merger algorithms have in common that they act locally on a specific node in a ring, without requiring global knowledge about the entire ring. Thereby each node forwards information, which is essential for merging the overlay, to an appropriate node, without violating the typical character of peer-to-peer applications: responsibility is distributed equally among all peers. Nevertheless, the performance of the merging algorithms differs clearly. Below, we analyze the introduced merging algorithms ([10] and [11]) and compare them to the Ring Reunion Algorithm. Therefore, in Section 3.2.1, we focus on the time complexity of the algorithms and in Section 3.2.2 we examine their message complexity. In Section 3.2.3 we examine the circumstances under which lookups are possible during a merging operation.

3.2.1 Time Complexity

For routing correctness only successor pointers have to be correct in a Chord ring. The finger entries are optional to increase Chord's lookup performance. An efficient merger algorithm therefore should be faster than Chord's stabilize protocol in combination with a simple join-leave approach, like the one introduced at the beginning of this chapter.

Chord-Zip

In Chord-Zip a lookup is necessary to start a merger instance. The token is then passed through the ring by all nodes. Therefore the complexity of the execution time of the Chord-Zip Algorithm is $O(N + log\,N)$, if no messages are duplicated and N is the number of all involved nodes.

Ring Unification Algorithm

The Ring Unification Algorithm also has to merge all N nodes. To achieve this, *MLookup*s are started frequently, which perform like normal Chord lookups. As a result the total execution time complexity of the Ring Unification Algorithm is $O(N \cdot logN)$. Still, two minor performance reductions can be found in the Ring Unification Algorithm. The closest preceding finger method which is used in *MLookup* (Listing 2.2) always forwards a search in clockwise direction, even if nodes are merged anti-clockwise (consider lines 22-25 in Listing 2.2 and Figure 3.7). Additionally, it might occur that few nodes are missed to be merged.

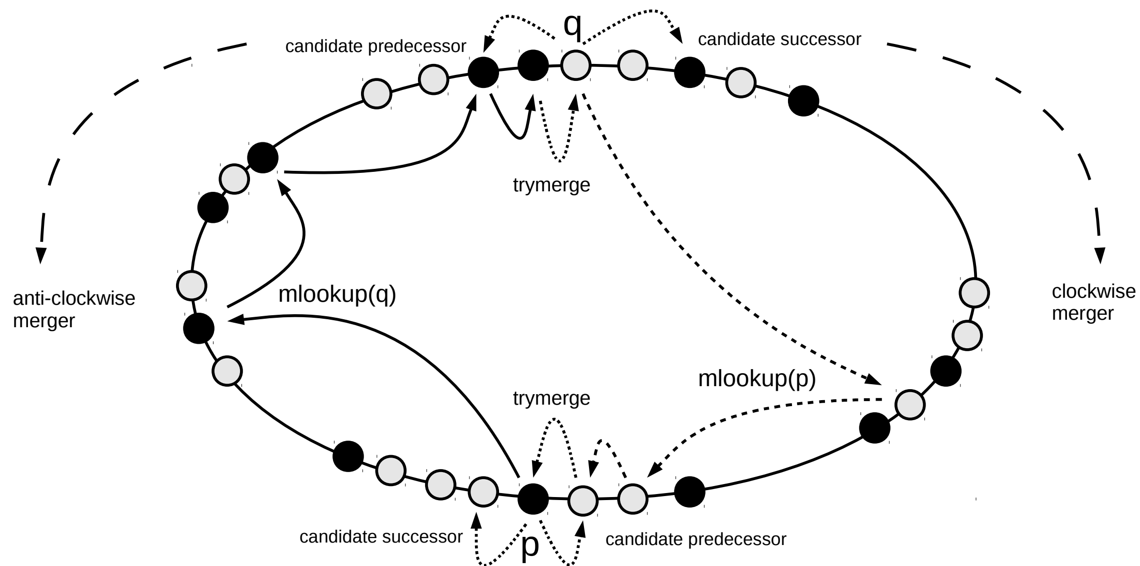


Figure 3.7: Ring Unification Algorithm operates in both directions: clockwise and anti-clockwise.

Ring Reunion Algorithm

For the Ring Reunion Algorithm an initial lookup is needed to start the merger. After that, the merger token is passed through the ring, once for each node until the algorithm terminates. Consequently, the time complexity of our algorithm is given by $O(N + log N)$.

3.2.2 Message Complexity

In order to increase performance and to avoid costly bandwidth consumption, a merging algorithm should operate quickly and produce as few messages as possible. Ideally the number of forwarded messages depends on the number of constructs which arise during a merger, rather than on the number of participating nodes.

Chord-Zip

In Chord-Zip every node sends a Zip-Ping message to its alternative successor pointer and receives a Zip-Pong message in turn. Besides, we assume that each initiator node starts a merger algorithm as soon as it receives a suitable contact node. Therefore Chord-Zip forwards approximately twice as many messages as nodes exist, which are supposed to be merged.

Ring Unification Algorithm

The Ring Unification Algorithm probably needs more messages to merge multiple rings than the other algorithms, since this algorithm starts kind of a lookup for every node which should be merged. Therefore $O(N \cdot logN)$ messages are produced during a merger, where N is the total number of participating nodes. Against that, the Ring Unification Algorithm starts to merge instances simultaneously, one to operate clockwise and another one to operate counter-clockwise. Besides, the algorithm is only started if an alive node is found within the passive list, so that the number of started merger instances is reduced reasonably.

Ring Reunion Algorithm

With the Ring Reunion Algorithm a merger token is forwarded by each participating node once. Thus the quantity of messages correlates to the number of nodes that are merged in general. If the Ring Reunion Algorithm is used in combination with the passive list, the number of merger instances relates to the number of nodes which are found to be alive. If instead, the *alive contact list*, with a specific number of contact nodes, is iterated periodically by each node to find new contact nodes, and a probability with which every node starts a merger instance is used, the message overhead is further reduced. With this method it is possible to start a fixed number of instances per construct, averagely, so that the bandwidth consumption does not depend on the number of nodes, but on the current number of constructs.

3.2.3 Lookups During Merger

As mentioned earlier, Chord's lookup performance bases on its finger entries, which have to be updated frequently to unfold Chord's full potential. In [12] the authors of chord demonstrate that if a stable Chord ring with N nodes and correct finger pointers is given, and "another set of up to N nodes joins the network, and all successor pointers (but perhaps not all finger pointers) are correct, then lookups will still take $O(log N)$ time with high probability", (Theorem IV:4 in [12]).

Lookups in Chord are forwarded via the method *closest_preceding_node()* until a finger is reached, which is the closest preceding node of the target id, without being responsible for it. Actually, to reach the node which is responsible for the target id, the closest preceding finger forwards the lookup to its successor (see *find_successor()* in Listing 3.3). This behavior ensures that lookups are always resolved by a successor pointer as last step, in case a finger entry might be corrupt or not close enough to the target id. If lookups are started then, they will be forwarded to the closest finger and not more

than as much nodes have to be visited as nodes had been merged. Therefore lookups will still take $O(log\,N)$ time after the merger. For this reason, no finger entries are combined in the Ring Reunion Algorithm, since this is not necessary, nor will this feature increase performance. Another reason not to copy this concept from the Chord-Zip algorithm is that the combination of finger entries never yields correct values for all finger entries.

Listing 3.3: Basic Chord methods.

```
n.create()
    predecessor = nil
    successor = n

n.join(n')
    predecessor = nil
    successor = n'.find_successor(n)

n.find_successor(id)
    if id ∈ (n, successor]
        return successor
    else
        n' = closest_preceding_node(id)
        return n'.find_successor(id)

n.closest_preceding_node(id)
    for i = m downto 1
        if finger[i] ∈ (n, id)
            return finger[i]
    return n
```

Listing 3.4: Chord's methods for stabilization.

```
// called periodically
n.stabilize()
    x = successor.predecessor
    if x ∈ (n, successor)
        successor = x
    successor.notify(n)

n.notify(n')
    if predecessor is nil or n' ∈ (predecessor, n)
        predecessor = n'

// called periodically
n.fix_fingers()
    next = next + 1
    if next > m
        next = 1
    finger[next] = find_successor(n + 2^(next-1))

// called periodically
n.check_predecessor()
    if predecessor has failed
    predecessor = nil
```

The combination of finger entries should not be done for on last reason. The authors of [23] criticize that ring-based overlays would not be able to perform lookups during a merger, even lookups addressed to the own ring would fail. If finger entries are combined then, like in Chord-Zip, it is not comprehensible, from which ring a single finger entry originates. If then a lookup is started during the Chord-Zip merger, it is not obvious in which ring the lookup will be forwarded. As soon as the lookup is forwarded by a node which has already combined its finger entries, the lookup is passed into one of the two involved rings randomly, depending on the finger entry, which has been chosen as better candidate. In Figure 3.8 for example, node n in ring R_1 starts a lookup for t. Depending on how node p has combined the gathered routing information, the lookup is forwarded within the own ring (top) or into ring R_2 (bottom), because node k has been considered to be closer (than s) to node t now.

If the finger entries are not combined, but rather adjusted by the Chord stabilize protocol, it is possible that lookups are successful during the merger. Either the lookup is passed to a part of the ring which has not been merged yet, or the lookup stops in a part of the ring which had been merged already. In the first case the finger entries in nodes that have not been merged yet have not changed, so that the target id is directly reachable via the closest preceding finger and the following sequence of successors. In

Figure 3.9 node n's requests are forwarded within ring R_1 as finger references have not been updated yet. Finger entries are only updated in parts of both rings that have already been merged.

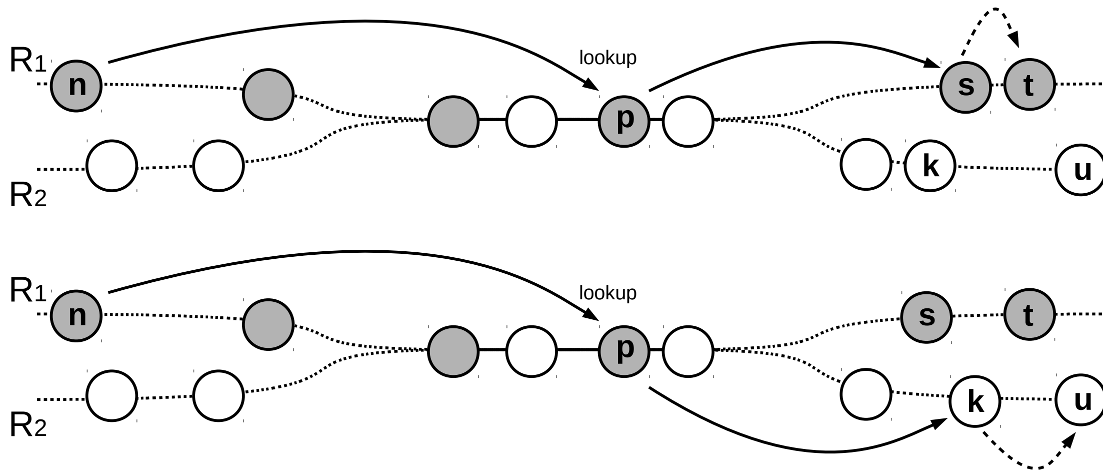


Figure 3.8: Lookup during merger after routing information has been combined.

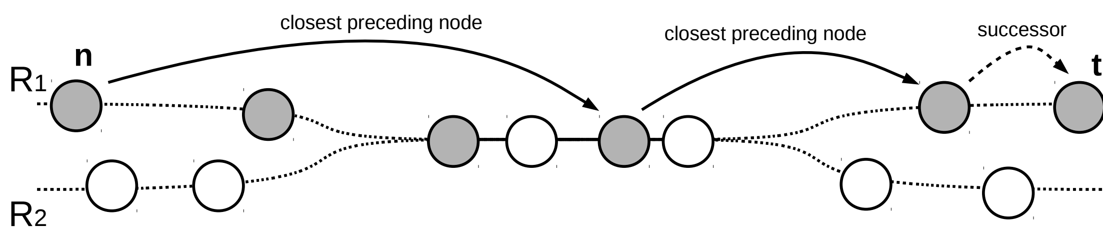


Figure 3.9: Successful lookup during merger.

Similar to Theorem IV:4 in [12], during a merger, each lookup is forwarded closer to the target id, and with high probability only O(log N) successors have to be traversed until the responsible node is found, since all nodes are distributed equally in Chord.

It might happen though, that Chord's stabilize protocol modifies finger entries during the merger. In that case, Chord searches a suitable successor for a finger entry via a normal Chord lookup, which uses the *closest_preceding_node()* method. Thereupon, if the closest preceding finger has been found, all successors are traversed, until a node among these successors is found, which fits better as finger entry. Consequently, a finger entry is replaced by a new one if, and only if a better fitting successor exists for this entry after the merger.

In case the lookup is addressed to a part of the ring which has not been merged by then, the lookup will not leave the initiator node's ring, as no better finger entries are known for this part of the ring. However, if a target id is placed in a part of the ring which has been merged already, it does not matter

if finger entries are updated, since this part of the ring can be seen as global ring now, and therefore Theorem IV from [12] holds. The assumption in [23] that Chord was not able to execute correct lookups during a merger can be refused.

Chapter Conclusion

In this chapter we have presented a detailed description of our Ring Reunion Algorithm. How it discovers new contact nodes has been examined in Section 3.1.1: a list of well known contacts is iterated periodically in order to find reachable contact nodes. Traffic overhead is reduced by estimating the size of the overlay construct one node is currently in and by limiting the overhead to a fixed number of messages per overlay construct (Section 3.1.3). We have described the algorithm's ability to start parallelized instances autonomously in Section 3.1.4 and we have discussed the importance of the algorithm's termination in Section 3.1.5. An accurate implementation of the Ring Reunion Algorithm in pseudo code has been proposed in Listing 3.2.

In Section 3.2.1, we have focused on the time complexity of the presented algorithms and in Section 3.2.2 we have compared the message complexity of the Ring Reunion Algorithm to the other approaches. Additionally, we have analyzed the *active contact list* which is iterated periodically to find new contact nodes and which calculates a probability with which merger instances are started. With this method the bandwidth consumption does not depend on the number of nodes, but on the current number of overlay constructs. In Section 3.2.3 we have demonstrated that lookups in Chord are possible during a merger procedure.

In Chapter 4 we describe the simulation environment which has been used for our simulations, we briefly describe the implementation of the merger algorithms, and we describe metrics which are used to judge the quality of the presented merger algorithms.

Chapter 4

Implementation

In the previous chapters the most important criteria have been summarized, which should be considered when a merging algorithm is designed. For this purpose, the Chord-Zip Algorithm, the Ring Unification Algorithm and our newly designed Ring Reunion Algorithm have been compared with each other. In this chapter we focus on the implementation of the Ring Reunion Algorithm. Therefore, in Section 4.1 we give an overview on the simulation environment which has been used for our simulations. A brief summary about the implementation of the merging algorithms we evaluate in Chapter 5 is given in Section 4.2. The most important metrics which are analyzed in our simulations are described in Section 4.3. Within the last two decades many network simulators have been developed and established, like ns-2 [25], ns-3 [26], OverSim [27] in combination with OMNeT++ [28], etc.. Since we are looking forward to test the merger algorithms with Chord, the chosen simulation engine should be able to simulate peer-to-peer systems. Although ns-2 and ns-3 are both well known and provide realistic simulations in many ways, both simulators do not offer any possibility to test peer-to-peer overlays. Therefore PeerfactSim.KOM [29] has been chosen for our simulations, because this simulator is easy to use and provides multiple possibilities to simulate peer-to-peer overlays and applications.

4.1 PeerfactSim.KOM

The PeerfactSim.KOM simulator has been developed at the TU Darmstadt at the Multimedia Communications Lab (KOM). Since 2011, the simulator has been extended and maintained at the University of Paderborn (UPB) and the Heinrich Heine University of Düsseldorf (HHU). The concept of the simulator is related to the ISO/OSI reference model and should be well structured and usable for a large variety of scenarios. Primarily, multiple peer-to-peer scenarios are supported without being dependent of a specific peer-to-peer architecture. Therefore the simulator is based on a plug-in concept and consists of different components which can be combined and extended for simulating any use case.

4.1.1 Components

The simulator's main component is the *Simulation Engine*, which is the core of the discrete event-based simulator. The *Simulation Engine* therefore contains important parts like the *Event Scheduler* or the *Event Queue*, which both in combination regulate the execution of pending tasks. Further, the *Simulation Engine* coordinates the other components of the simulator. During a simulation, each peer is represented by a host, which comprises different ISO/OSI layers and holds information about its state, for example upload and download bandwidth, consumed bandwidth, CPU power, storage, etc.. The *Event Scheduler* distributes all simulated events to the different layers of the hosts, which react on those events hereupon. Finally the *Simulation Engine* logs different outputs and provides a GUI to visualize executed simulations.

Above the *Simulation Engine* different layers are located, which can be configured independently of each other and combined according to the users wish, to cover as many scenarios as possible. Although the layers refer to the ISO/OSI reference model their design is focused on the simulation of peer-to-peer systems. The lowest layer is the network layer, which is connected to a component called *subnet* and to the transport layer inside each simulated host. The *subnet* component connects all hosts with each other and simulates traffic between individual hosts via the Internet. The transport layer is located on top of the network layer to provide end-to-end communication services on higher layers. The most important layer for peer-to-peer simulations might be the overlay layer. PeerfactSim.Kom comes with an already implemented selection of interesting overlays like Chord [12], CAN [15], Kademlia [14], Gnutella [30], etc.. In addition, the application layer allows the user to run different applications on top of a selected overlay. Either the user configures a simulation to use an already implemented application like a file-sharing application or a lookup generator, or the user creates his or her own application which can easily be integrated into existing implementations.

For evaluations of individual simulations it might be useful or even necessary, to obtain specific data and information, which are produced during a simulation. PeerfactSim.KOM does not provide any mechanism to obtain all possible kinds of data, instead it comprises a monitoring architecture, which has access to different fundamental information. Furthermore, multiple analyzers are implemented, which already give access to basic information like bandwidth, number of online hosts, sent and received messages and much more. Directly after a simulation, this collected data is plotted with *gnuplot*. Besides, it is possible that developers implement own analyzers and integrate them into PeerfactSim.KOM to obtain specific data, which is not collected by the simulator by default. For this purpose the Java class *org.peerfact.impl.util.oracle.GlobalOracle* can be used, which gives access to all hosts during a simulation. In Addition the simulator provides abstract analyzer classes, which are helpful to gather data during a simulation and plot them afterwards.

Listing 4.1: Example of a XML configuration file.

```
1  <?xml version='1.0' encoding='utf-8'?>
2  <Configuration>
3    <Default>
4      <Variable name="seed" value="0" />
5      <Variable name="finishTime" value="60m" />
6    </Default>
7
8    <SimulatorCore class="org.peerfact.impl.simengine.Simulator"
9      static="getInstance" seed="$seed" finishAt="$finishTime">
10   </SimulatorCore>
11
12   <NetLayer class="org.peerfact.impl.network.simple.SimpleNetFactory">
13     <LatencyModel
14       class="org.peerfact.impl.network.simple.SimpleStaticLatencyModel" />
15   </NetLayer>
16
17   <TransLayer
18     class="org.peerfact.impl.transport.DefaultTransLayerFactory" />
19
20   <Overlay
21     class="org.peerfact.impl.overlay.dht.chord.chord.components.ChordNodeFactory" />
22
23   <Monitor class="org.peerfact.impl.common.DefaultMonitor"
24     start="0" stop="$finishTime">
25     <Analyzer
26       class="org.peerfact.impl.overlay.dht.chord.base.analyzer.ChordStructureAnalyzer" />
27   </Monitor>
28
29   <HostBuilder
30     class="org.peerfact.impl.scenario.DefaultHostBuilder"
31     experimentSize="30">
32     <Group groupID="Germany" size="30">
33       <NetLayer />
34       <TransLayer />
35       <Overlay />
36     </Group>
37   </HostBuilder>
38
39   <Scenario class="org.peerfact.impl.scenario.CSVScenarioFactory"
40     actionsFile="chord-actions.dat"
41     componentClass="org.peerfact.impl.overlay.dht.chord.chord.components.ChordNode">
42   </Scenario>
43 </Configuration>
```

4.1.2 Simulations

Scenarios in PeerfactSim.KOM are defined by configuration files, which are represented by a structured XML file. Each configuration file contains the components of the simulator which are supposed to be used for a given scenario. Listing 4.1 shows an example of such a configuration file. Within

the given file the *Simulation Engine*, a simple network layer, the default transport layer and the Chord overlay are used. Furthermore the usage of an Chord structure analyzer and the number of given hosts is determined. In lines 40 and 41, the scenario is extended by determining the class of the used overlay and the reference to an action file that is executed during simulation. This action file may contain commands that are executed by the component class at a given time. *Germany 1m-8m join callback* for example (stored in chord-actions.dat) tells all simulated hosts from group Germany to join the overlay which is chosen by the component class from minute 1 to minute 8.

4.2 Implementation

As mentioned previously in Chapter 3, we consider ring-based overlays in this work, as they are very popular and robust against failures. Therefore the merging algorithms which are supposed to be simulated have been implemented directly into the existing Chord implementation of the simulator. Each merging algorithm has been implemented in a separate Java class, so that each merger could easily be used in combination with other ring-based overlays, after little adoptions. In PeerfactSim.KOM it is possible to pass any parameter to any simulated Java class via the XML configuration script. We modified the class *org.peerfact.impl.scenario.CSVScenarioFactory*, so that it is possible to start a simulation with different merger algorithms and parameters, which are plotted and compared with each other directly. Mainly each merger process consists of two parts: the *contact list* and the *merger*.

4.2.1 Contact List

The *contact list* is the method which is necessary to find a contact node in another ring. Hence the *contact list* searches for nodes which can be used to initiate a merger algorithm. In this work we consider three different methods that can be used to find a contact node.

- Start merger *manually*
- Discover contact nodes via *passive list*
- Discover contact nodes via *active contact list*

Manually

This method is the simplest way to find a contact node: a specific user or an administrator can pass a contact node to the initiator node manually, so that a merger instance is started hereupon. For this purpose we extended the class *org.peerfact.impl.overlay.dht.chord.base.components.ChordBootstrapManager*

with the method *getRandomAvailableGroupContact()*, which returns a random contact node from a specific group. It is possible then to start each merging algorithm manually via the command *merge-Group* out of a scenario action file.

Passive List

The idea of the passive list has been introduced in [11] by Shafaat et al. and has been implemented in this work. Every time a message transmission fails, the contact information of the receiving node is passed to the sender's passive list, which tries to get in contact with each failed node periodically. Nodes which are detected to be alive again (by using ping-pong messages) are instantly handed over to the chosen merger algorithm. Nodes which do not respond to a ping message, are further kept in the passive list.

Active Contact List

Another method to find suitable contact nodes is to select a random node from a list of possible contact nodes. Such a list for example could be passed to a specific node manually by a system administrator. Another option would be to obtain such a list from the bootstrap node, which is used by each node to join the overlay. Additionally, each node could execute IP-scans frequently to find alive nodes and to update its *active contact list*. As a last option multiple servers could store lists of participants and give parts of those lists to requesting nodes. In this thesis the *ChordBootstrapManager* which is provided by PeerfactSim.KOM is used to obtain an initial contact list. Periodically, each node selects a node form the list, which is considered then as contact node candidate. Ere this contact node is contacted and merged, the *active contact list* calculates $\alpha/size_c$, the probability with which the *active contact list* passes the newly found contact node to the merger. Therefore approximately α nodes per overlay construct with size $size_c$, start a merger algorithm each period. If then one node is permitted to merge the contact node, it sends a ping message to the contact node. Only if a pong message is received within a small time frame, the *active contact list* hands the contact node over to the appropriate merger algorithm.

4.2.2 Merger

The merger is the most important part of the merging process. The chosen merger algorithm obtains a contact node from the respective contact list and tries to merge this contact node hereafter. Since the above mentioned merger algorithms operate in a similar way the idea of the *detqueue* in combination

with the passive list can be used by any other merger to store available contact nodes. In combination with a *detqueue*, every merger class periodically starts a timer, on whose expiring a single contact node is taken from the *detqueue* and merged in addition.

Using the Chord-Zip Algorithm or the Ring Reunion Algorithm it is necessary to start a lookup to obtain a suitable alternative successor. Both algorithms ask the contact node to search for the alternative successor and wait for its response. For this purpose we modified a simple *join operation*, so that either the alternative successor is given to the asking node, if the contact node is reachable, or the contact node is passed to the contact list (passive or active) again if it does not respond to the node's query. Similar to Chord-Zip, the Ring Reunion Algorithm can use retransmission optionally in case a *merge* message or a *distribution* message gets lost. The Ring Unification Algorithm does not need to perform an initial lookup, instead the contact node is directly merged, since successor and predecessor nodes are approached via *MLookup*.

If the Ring Reunion Algorithm is used, the *distribution* algorithm is executed directly after the alternative successor is found via *join operation*. If one node then receives a *distribution* message, it starts a *join operation* hereupon and distributes its knowledge to other nodes in its finger table, as long as the counter of the *distribution* message is less than a specific value. Finally the merging algorithm is started in order to unite multiple rings to a common ring.

4.2.3 Isolation Model

Currently, PeerfactSim.KOM is able to isolate different groups at a given time for a fixed duration. In order to enable the isolation of any region at any time and to disable the isolation of any region independently, we extended the simulator (especially *org.peerfact.impl.scenario.CSVScenarioFactory*), so that isolation events can be defined in every PeerfactSim.KOM action file. To isolate group Norway at minute 30 for example, one could write *Isolation Norway 30m disconnect* into the corresponding action file. *Isolation Norway 60m connect* on the other hand will undo the isolation again.

4.3 Metrics

To be able to compare the merger algorithms with each other, it is necessary to specify criteria which explain the performance and correctness of a single merger. First of all a merger should be able to merge multiple rings in a certain time, that means multiple rings should be combined in the way that one global ring is created as a result. Second, a merger should be capable of arranging all successor pointers in a proper way, since this is the most important criterion that routing within a ring is possible.

In order to gather this metrics, a new analyzer has been integrated into PeerfactSim.KOM by extending the class *org.peerfact.impl.analyzer.AbstractFileAnalyzer*. This new analyzer periodically checks in which way the successor pointers are related to each other, writes those information into a file and plots the results afterwards.

4.3.1 Number of Constructs

With this metric, all successor pointers are investigated and the current number of constructs is observed, which is represented by the way the successors are arranged. Consequently, we distinguish between three different constructs: full circles, hanger ons and chains (see Figure 4.1).

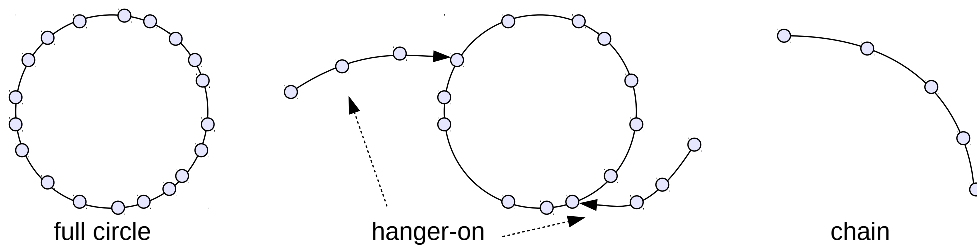


Figure 4.1: Possible overlay constructs which arise during a merging process.

To ascertain the number and kinds of constructs, information about peers and their successor pointers, especially the topology they describe, are copied into an array by the analyzer. Each connection between peers and successors is then traversed in order to determine if a full circle exists. Within a second iteration the analyzer examines if some nodes form a chain which is freestanding (chain) or are attached to a full circle (hanger-on). With this information it is easy to decide if a single global ring is created or not, but it is not possible to determine the number of correct pointers with this information. For example: two constructs could stand for two independent rings or a full circle and a single node that is attached to the ring. The statement that more than one construct exists is not accurate enough to evaluate the performance of a merger algorithm.

4.3.2 Correct Pointers

For better evaluation of any merger algorithm we have to determine the number of correct successor pointer at a specific time. Therefore we extended the analyzer in the way that each successor pointer is periodically compared with the value it should contain in a single global ring. Consequently, this metric describes perfectly the fraction of current correct pointers in comparison to overall correct pointers.

Chapter Conclusion

In this chapter we have focused on the implementation of the merging algorithms. In Section 4.1 we have therefore given a brief overview on the simulation environment which has been used for our simulations. We have explained how merging algorithms obtain new contact nodes (4.2.1) and how those algorithms operate (4.2.2). The most important metrics, which are used in our simulations to rate the quality of the different merger algorithms, have been described in Section 4.3. We use the number of overlay constructs as a metric, to determine roughly the point in time at which a single overlay construct is formed. Since this metric is not precise enough, we determine the number of correct successor pointers at a specific time in order to determine the operational time of any merger algorithm accurately.

In the next chapter we describe simulation setups which are examined to judge the presented merger algorithms. In Section 5.2 of Chapter 5 we present the results of our simulations and point out why the Ring Reunion Algorithm outperforms the other merger approaches.

Chapter 5

Evaluation

In this chapter we evaluate the behavior of each merging algorithm in order to obtain insights on the quality of the different merging mechanisms. The most important criterion to be verified is whether a merging algorithm is able to unite a disrupted overlay or not and which mechanisms and properties of one algorithm are necessary to do so. Secondly, we try to identify mechanisms which help further to increase the mergers' performance. Thus, we focus in our evaluation on the following metrics:

- Periodically, all successor pointers are investigated in order to determine the number of overlay constructs that are formed during our simulations. We distinguish between three different constructs which might arise: full circles, hanger-ons and chains. Using the number of overlay constructs as a metric allows us to determine the point in time at which only one construct is created. Therefore, we can easily decide whether one merger algorithm is able to unify all network partitions to a single construct.

- For better evaluation of the presented merger algorithms, we determine the number of correct successor pointers at a specific time, as the number of constructs is not accurate enough. With this metric it is possible to determine the operational time of any merger algorithm precisely. In our evaluation the number of correct successor pointers is presented as fraction $\frac{P_{current}}{P_{global}}$, where $P_{current}$ denotes the number of all successor pointers which are currently correct and P_{global} denotes the number of correct successor pointers in a single global ring, i.e the number of all nodes in a scenario.

- In order to rate the quality of the merger algorithms, we focus on the mergers' traffic overhead in terms of messages and bandwidth consumption. Ideally, the produced messages of any merger algorithm will relate to the number of constructs in the underlying network.

Below, in Section 5.1 we will describe the setups of our simulations. In Section 5.2 we are going to discuss the results of each simulation. Thus, time designations below refer to simulated time.

5.1 Scenario Setups

For our simulations we use the event-based peer-to-peer simulator PeerfactSim.KOM in order to obtain realistic results and insights on our research. As described in the previous chapter, all our simulations are based on the Chord overlay which has been implemented in the simulator already. Every simulation has been run with 10 different random seeds, so that all values in the graphs represent the average of 10 different values. Each algorithm has been tested separately and independently from other merging algorithms. In addition, each simulation uses GNP coordinates [31] to describe delays in the fundamental network realistically. Reasonable approximations for jitter and message delays are integrated into the simulator by using measurements from the PingEr project [32].

In most of our simulations we do not consider churn and packet loss, as both attributes might obscure the characteristic behavior of a specific merging algorithm. First, in Section 5.1.1, we investigate scenarios in which the merging process is manually initiated in order to measure the operational time of each merging algorithm. Next, in Section 5.1.2, we enable automatic partition detection and merging to simulate realistic cases.

5.1.1 Merging Manually

In the scenarios, which are listed below, only one contact node per Chord ring is given manually to a suitable initiator node. Hence, only one merger instance per Chord ring is started. We investigate in scenario Setups A.1, A.2 and A.3 the merging quality of Chord-Zip, the Ring Unification Algorithm and our Ring Reunion Algorithm. Setup B.1 constitutes a direct comparison of the performance of the mechanisms Ring Unification and Ring Reunion.

A.1: Merging Two Rings

Within the first scenario we examine the Chord-Zip Algorithm, the simple Ring Unification Algorithm, the gossip-based Ring Unification Algorithm, the Ring Reunion Algorithm and the Ring Reunion Algorithm with four parallel instances. Further, we tested Chord's behavior without any merger algorithm executed and Chord's behavior if the initiator node sets its successor pointer directly to the contact node. This scenario should reveal if a specific algorithm is capable of unifying two separated Chord rings and give a first feeling for how each algorithm performs under simple conditions. As described above each algorithm has separately been simulated with 10 random seeds, message delay and jitter. Packet loss has not been enabled to get better insights on the basic properties of each algorithm. From the beginning of the simulation to the 150th minute, all of the 1024 participating nodes join one

of two Chord rings, so that after 150 minutes two rings with 512 nodes in each ring are created. In the 180th minute the merger algorithms are started manually. To ascertain that exactly one merger instance is started in each simulation, one contact node is given (via scenario action script) to another node from the other ring respectively. This scenario is similar to the case in which an administrator gives a contact node to a specific node to initialize a merger. The duration of the whole simulation is limited to 360 minutes, so that each merger algorithm should unite the two rings within the remaining 180 minutes.

A.2: Merging Three Rings

Within the next scenario we investigate the Chord-Zip Algorithm's, the simple Ring Unification Algorithm's, the gossip-based Ring Unification Algorithm's and the Ring Reunion Algorithm's (1 and 4 instances) ability to handle multiple merger instances simultaneously. As we see later, this property turns out to be important if multiple instances are started automatically whenever a contact node is detected, because in such a case it might happen that one contact node is supposed to be merged by two initiator nodes from different rings at the same time. Although this problem might occur every time when two regions are connected again after an isolation phase, it must not hinder the merging algorithm to terminate before, or long time after, a global ring is created. In its current version the PeerfactSim.KOM simulator does not provide the functionality to form multiple Chord rings directly. Therefore we extended it with the ability to form groups directly via action script. At the beginning of this scenario three different Chord rings are created with 341, 341 and 342 nodes respectively. Within the 10th minute, two nodes, each selected from one of the two rings with 341 nodes, start to merge one and the same contact node from the 342-node ring. Again, the contact node is given to the initiator nodes via the scenario action script. After 180 minutes the simulation is finished.

A.3: Merging Five Rings

In the following, we extend the simple scenario in which two nodes start to merge one and the same node simultaneously, to study whether multiple rings can be merged within a more complex situation. Further, we would like to know if it is possible to merge two rings, even if both are merged by other rings concurrently. This time we simulate the simple Ring Unification Algorithm, the gossip-based Ring Unification Algorithm, the Ring Reunion Algorithm and the Ring Reunion Algorithm with four parallel instances, again 10 random seeds per algorithm. Unfortunately the Chord-Zip Algorithm fails to merge three rings at the same time, since it does not know when to stop its instances. Thus, we do not consider the Chord-Zip Algorithm in this scenario. At the very beginning of each simulation five rings are formed with 160, 231,271, 205, 157 nodes, denoted as group Latin America, Germany,

Florida, Czech Republic and South Africa respectively (see Figure 5.1). In the 10th minute one node from South Africa starts to merge a node from group Latin America. Hereafter, in the 60th minute one node from Florida starts to merge a random node from Latin America, while one German node starts to merge one node from group Florida. Meanwhile one node from the Czech Republic starts to merge a contact node located in Latin America as well. The simulation is again stopped after 180 minutes.

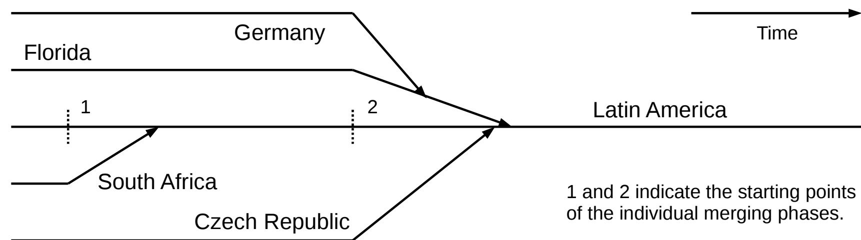


Figure 5.1: Scenario in which five groups are merged.

B.1 Comparison of Performance

Since it was found that the gossip-based Ring Unification Algorithm, which starts four additional instances in our simulations, performs similar to the Ring Reunion Algorithm, we found it interesting to compare the algorithms' performance during the unification of two Chord rings with 5121 nodes in each ring. Therefore two Chord rings are formed at the beginning of each simulation. In the 10th minute a contact node from one group is, again manually via action script, given to the initiator node which is located in the other group. The simulation is finished after 180 minutes. The gossip-based Ring Unification Algorithm is tested with a fanout parameter of 4, since this value is suggested by the algorithm's authors in [11]. The Ring Reunion Algorithm is executed in this scenario with 1, 4, 8 and 16 parallel instances.

5.1.2 Merging Automatically

The previous scenarios have in common that in each simulation only one contact node is manually given to another node, so that a merging instance is started. This simulation can be compared to a real situation in which a network administrator wants to merge two or more Chord rings. Within those simulations the methods which are used to detect alive contact nodes, as described in Section 4.2.1, are disabled, to get a feeling for the fundamental character of each merging algorithm. Below, the automatic searching techniques are enabled to study the algorithms' behavior if multiple instances are started in a realistic scenario. Further, our goal is to determine which algorithms are qualified to reunite overlay nodes independently after network failures.

C.1: Study of Passive List

First of all we investigate the passive list in combination with the Chord-Zip Algorithm, the simple Ring Unification Algorithm, the gossip-based Ring Unification Algorithm and the Ring Reunion Algorithm with one instance and four parallel instances. Every algorithm has been executed in 10 simulations with different random seeds, message delay and jitter. The passive list has been configured in the way that every 3 minutes an active node, if one is found, is passed to the respective merger algorithm, which then starts the merging process. At the beginning of the simulation all nodes join a common Chord ring. After 150 minutes 1024 nodes have joined a global Chord ring and the joining phase ends. In the 180th minute a group of 310 nodes is, due to an isolation event, separated from the other nodes. During the isolation, nodes in both separated regions search for other reachable nodes. With high probability multiple groups are formed directly after the isolation starts (see Section 1.2.2). During this time frame, all nodes that reach other contact nodes start merger instances. After one hour, i.e. in the 240th minute, the isolation is canceled so that all nodes are reachable by other nodes again. After 360 minutes the simulation is stopped.

The same scenario has been examined with the usage of a random contact list. Using this method, every node periodically selects a random node out of a list which contains all simulated nodes. In addition each node estimates the size of the construct it is currently in and starts a merger instance with probability $1/size_c$, where $size_c$ is the estimated number of nodes in the current overlay construct, so that only one merger instance per construct is created on average.

C.2: Study of Active Contact List

Since in a realistic scenario not all nodes are directly accessible or even known by other participants, the random contact list can not be used effectively in real life. Therefore we limited the random selection to a small quantity of 160 contact nodes, which are randomly chosen by the bootstrap node and given to each node that joins the overlay. With this scenario we try to find out if such an *active contact list* of 160 random contact nodes is sufficient to reunite a disrupted Chord ring. We are confident that this is possible, since all nodes are equally distributed among the random lists so that every node at least knows one node which is located in another region or knows a participant which knows such a node. The simulation begins with a join phase in which 1024 nodes join a common Chord ring. During a join operation each node asks its bootstrap node to deliver a list of 160 random nodes, which is stored by the nodes then. After 150 minutes the join phase is finished. In 240-second intervals each node checks if the next contact node on the *active contact list* is reachable. If this is the case, it starts to merge the contact node, even if it is located in the own ring. If all 160 nodes are inspected, the node starts to contact the first node on the *active contact list* again. In the 180th

minute, a group of 310 nodes is separated from the other nodes. During this isolation phase, groups within the same region are able to form a common ring, since the merging algorithm ensures it. After 60 minutes, i.e. in the 240th minute, the isolation is suspended so that the 310 nodes are reachable again. After 360 minutes the simulation is finished. With this scenario we examine the gossip-based Ring Unification Algorithm and the Ring Reunion Algorithm (1 and 4 instances). 10 random seeds for each algorithm, jitter and message delay have been simulated. The Chord-Zip Algorithm has not been considered, since it already fails to merge more than two rings simultaneously.

C.3-5: Study of Probability

The previously introduced *active contact list* is now tested in combination with a probability which is concerned by each node before it initiates a merger instance. With this scenario we examine the gossip-based Ring Unification Algorithm and the Ring Reunion Algorithm (1 and 4 instances), again with 10 seeds per simulation, message delay and jitter. Again, every 240 seconds the *active contact list* selects a possible contact node. Before it is merged though, each node estimates the size of the construct it is part of. Hereupon the node picks a random value and initiates the merger only if this value is below or equals $\alpha/size_c$, where α is 1, 5, 10 or 100 in this simulation and $size_c$ is the estimated number of nodes in the current construct. By this means, only 1, 5, 10 or 100 merger instances are started per construct every 240 seconds on average. Similar to the previous scenario a Chord ring with 1024 participants is formed from minute 0 to minute 150. After 180 minutes a group of 400 nodes is isolated, after 240 minutes it is connected again to the remaining groups and after 360 minutes the simulation is finished.

D.1: Complex and Realistic Scenario

This scenario represents a complex and more realistic scenario in which multiple regions are isolated. Again we examine the *active contact list* in combination with the gossip-based Ring Unification Algorithm and the Ring Reunion Algorithm (1 and 4 instances) with 10 random seeds per simulation, plus message delay and jitter. As described in the previous scenario setup, 1024 nodes join a common Chord ring during the first 150 minutes. A group of 400 nodes is isolated from the 180th minute to the 240th minute. In addition another group of 50 nodes is isolated from the 200th minute to the 240th minute. From minutes 240 to 300 a third group of 100 nodes is isolated from the other regions. To reduce the quantity of messages sent by the merging algorithms each node only starts a merger if it picks a random value below or equals $\alpha/size_c$, where $size_c$ is the estimated number of nodes in the current construct. We examine this scenario with $\alpha = 1$ and $\alpha = 10$, to compare a poor value for α (1) to a fair one (10). The Simulation is finished after 360 minutes.

E.1-2: Parameter Studies and Churn

In the previous scenarios we did not consider churn in order to focus on the fundamental behavior of each tested algorithm. As a next step, we evaluate the Ring Reunion Algorithm in the presence of churn. Again our scenario begins with the joining phase in which 1024 participating nodes form a Chord ring. In the 180th minute, when all nodes have joined the network successfully, a group of 400 nodes is separated from the rest of the Chord ring. The duration of the isolation lasts 60 minutes. From the 240th minute to the 360th minute, when the simulation is finished, the nodes are given time to unify the partitioned network again. This time, churn is enabled throughout the whole simulation, to show that our Ring Reunion Algorithm is able to handle it without loss of performance. In addition with this last simulation setup we examine the influence of different parameter settings on the Ring Reunion Algorithm. In simulation Setup C.3 we investigate the influence of α on the performance of our merger and on the quantity of messages which is produced. Supplementary parameters we examine in this scenario are the parameter which regulates the number of parallelized merger instances and the interval within which the *active contact list* is iterated. We simulate the Ring Reunion Algorithm with 4,8,16 and 32 parallel instances, each algorithm in combination with $\alpha \in \{10, 100\}$. The interval with which the *active contact list* selects a merging candidate is set to 5 minutes. Again we simulate 10 random seeds for each combination. Furthermore we simulate the Ring Reunion Algorithm with 4,8,16 and 32 parallel instances, each combined with intervals of 5,10,15 and 20 minutes in which merger instances are started, in order to find a configuration which represents a good compromise between operation time and bandwidth consumption.

5.2 Simulation Results

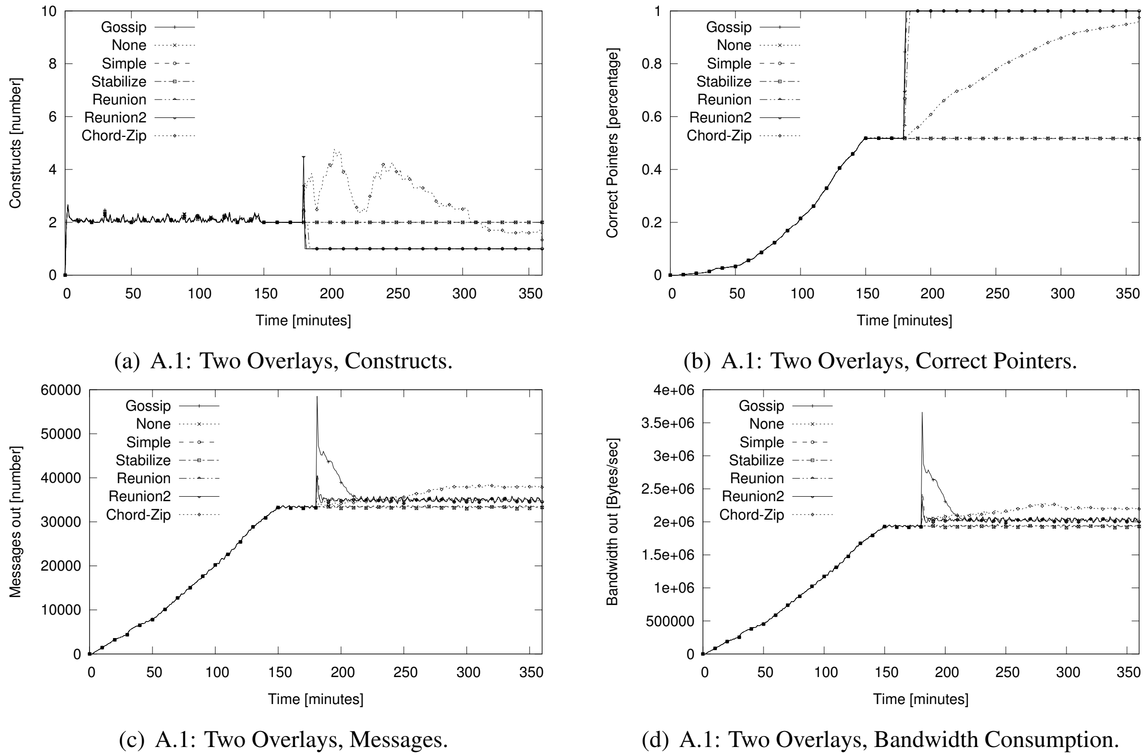
Below, we evaluate the simulation results which have been obtained by simulating the scenario setups from Section 5.1. In Sections 5.2.1, 5.2.2 and 5.2.3 we investigate Chord-Zip, the Ring Unification and the Ring Reunion Algorithm on networks with 2, 3 and 5 partitions. In order to compare the performance of the parallelized Ring Reunion Algorithm with the gossip-based Ring Unification Algorithm, we tested both algorithms with 10242 nodes. The results of this scenario are studied in Section 5.2.4. Next, in Sections 5.2.5, 5.2.6 and 5.2.7 we examine the merging algorithms in realistic scenarios with automatic partition detection enabled. In Section 5.2.8 we prove that our Ring Reunion Algorithm is able to unify network partitions even in complex cases in which multiple regions fail almost simultaneously. Finally, in Section 5.2.9, we evaluate the performance of our algorithm with the presence of churn and we suggest a fair configuration for our Ring Reunion Algorithm by evaluating the results of our simulations.

5.2.1 A.1: Merging Two Rings

The results of the first simulation are shown in Figure 5.2. Figure 5.2(a) shows that the stabilize protocol is not able to unify the two rings of 512 nodes by itself, although it was able to merge two disrupted rings in smaller scenarios with about thirty nodes. The stabilize protocol was started by setting one node's successor to a random node from the second ring. Furthermore Figures 5.2(a) and 5.2(b) reveal that both Ring Unification Algorithms and the Ring Reunion Algorithm are capable of merging two rings without major effort and in similar time. All successor pointers are corrected within a short period of time. In Figure 5.2(b) one can see that the Chord-Zip Algorithm needs more time to adjust the successor pointers. In 4 of 10 simulations the Chord-Zip Algorithm was successful to merge two rings within the simulated time. The reason why this algorithm is much slower than the other solutions is the way it choses its alternative successor pointer, which is always taken from the other ring. Because of this behavior the algorithm does not preserve the correct order of the nodes it merges (see Section 2.4.3). The number of messages sent by each algorithm each minute and the resulting bandwidth consumption are shown in Figures 5.2(c) and 5.2(d). It can be seen that after all nodes have joined the overlay, approximately 34000 messages per minute are sent by Chord to maintain both rings. The message overhead by the Ring Unification Algorithms and the Ring Reunion Algorithm, parallelized or not, is quite small. From the 250th minute to the end of the simulation it can be seen that the Chord-Zip Algorithm is sending approximately twice the amount of messages than the other algorithms, since all its Zip-Ping messages are answered with a Zip-Pong.

5.2.2 A.2: Merging Three Rings

Figure 5.3 shows the results of the scenario setup in which three different rings are merged simultaneously. In Figure 5.3(a) it can be seen that the simple and the gossip-based Ring Unification Algorithm, as well as the Ring Reunion Algorithm perform well, whereas the Chord-Zip Algorithm has its problems. Nevertheless, as Figure 5.3(b) reveals, the Chord-Zip Algorithm manages to adjust the successor pointers in a very slow tempo that would exceed the simulation time. Reason for this is the above mentioned choice of the alternative successors. Figures 5.3(c) and 5.3(d) indicate that the message and bandwidth consumption of all merger algorithms is similar, although the measurements of the Chord-Zip Algorithm fluctuate apparently around the other values.



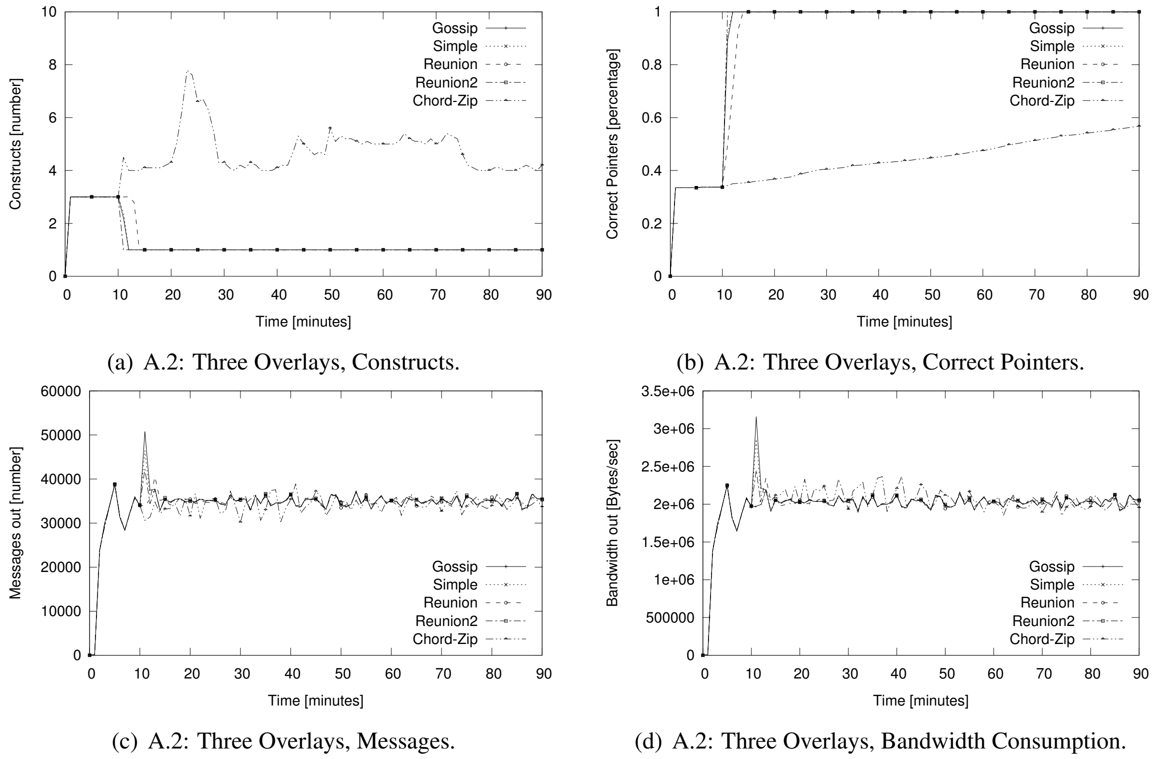
(a) A.1: Two Overlays, Constructs.

(b) A.1: Two Overlays, Correct Pointers.

(c) A.1: Two Overlays, Messages.

(d) A.1: Two Overlays, Bandwidth Consumption.

Figure 5.2: A.1: Merging of two networks.



(a) A.2: Three Overlays, Constructs.

(b) A.2: Three Overlays, Correct Pointers.

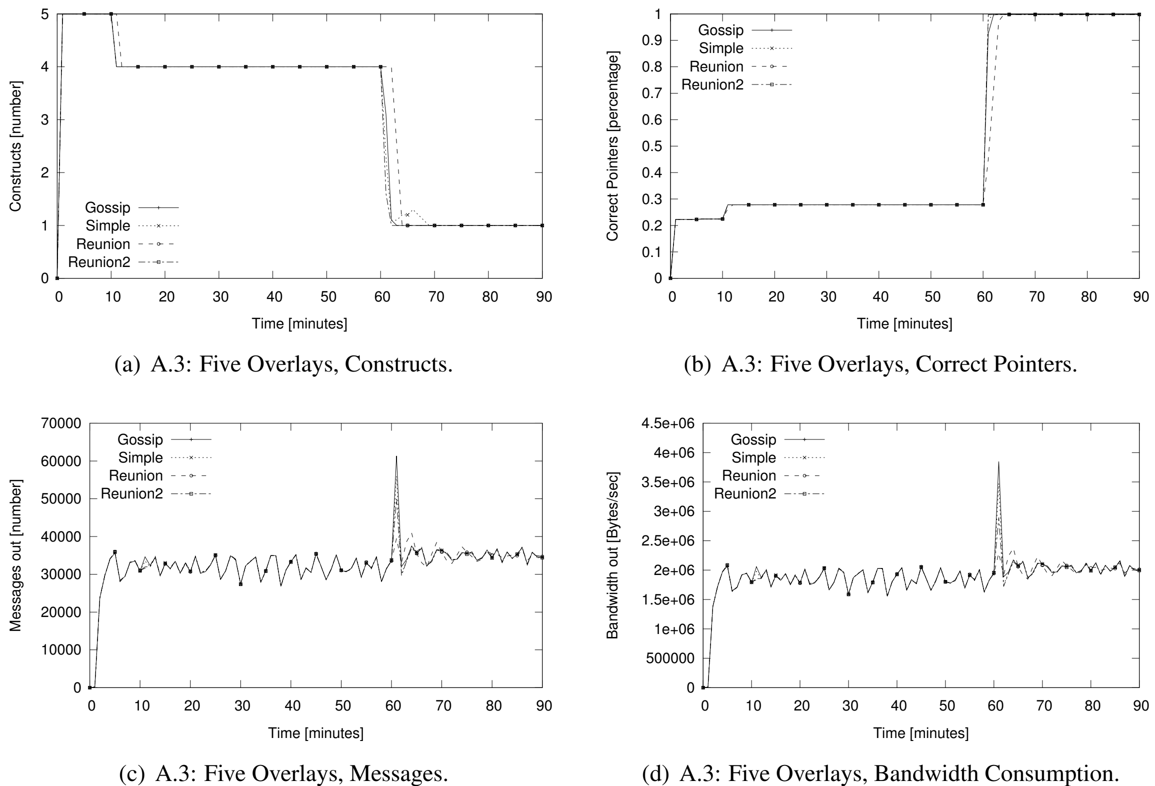
(c) A.2: Three Overlays, Messages.

(d) A.2: Three Overlays, Bandwidth Consumption.

Figure 5.3: A.2: Merging of three networks.

5.2.3 A.3: Merging Five Rings

In the following, we focus on Figures 5.4 and 5.5 which represent the merging of five different rings. In this scenario only the simple Ring Unification Algorithm, the gossip-based Ring Unification Algorithm, the Ring Reunion Algorithm and the Ring Reunion Algorithm with four parallel instances have been simulated, as the Chord-Zip Algorithm has turned out to be to slow. As one can see in Figure 5.4 all algorithms behave similar in merging time. It is noticeable that a peak occurs in the measurements of message and bandwidth consumption in Figures 5.4(c) and 5.4(d) at minute 60. This peak indicates the begin of the algorithms, which are integrated into the gossip-based Ring Unification Algorithm and the Ring Reunion Algorithm, to start multiple instances. Closer looks onto minute 60 reveal the performance of the single algorithms. In Figure 5.5(a) one can see that the Ring Reunion Algorithm, started with four parallel instances, is the fastest merger in this scenario, followed by the gossip-based Ring Unification Algorithm. The basic Ring Reunion Algorithm without any parallelization seems to be slowest algorithm at first, but in some cases the simple Ring Unification Algorithm has been slower.



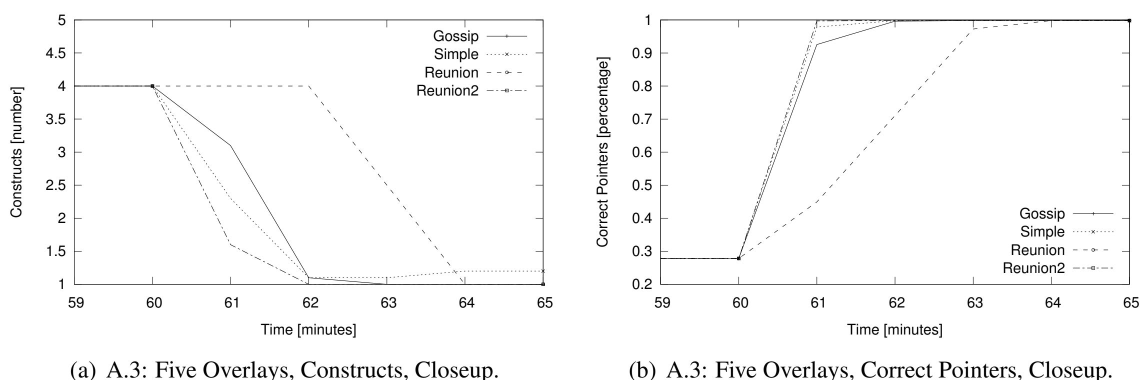
(a) A.3: Five Overlays, Constructs.

(b) A.3: Five Overlays, Correct Pointers.

(c) A.3: Five Overlays, Messages.

(d) A.3: Five Overlays, Bandwidth Consumption.

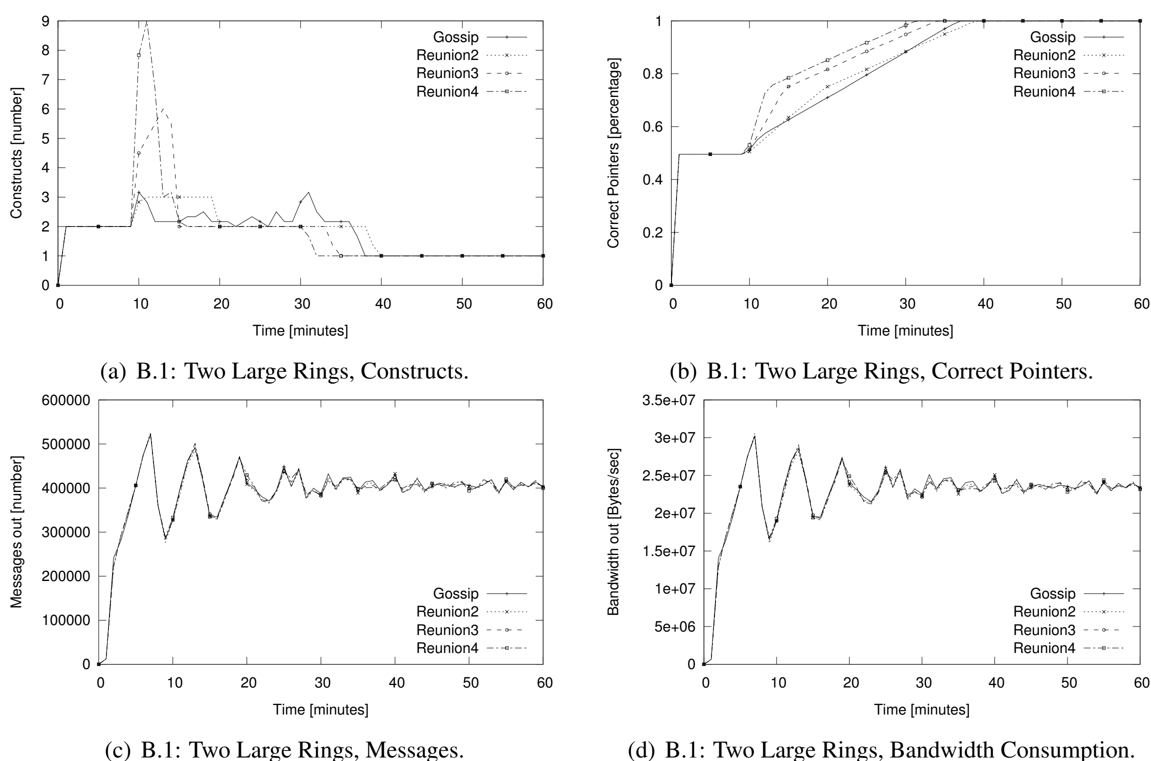
Figure 5.4: A.3: Merging of five networks.



(a) A.3: Five Overlays, Constructs, Closeup.

(b) A.3: Five Overlays, Correct Pointers, Closeup.

Figure 5.5: A.3: Closeup on merging of five rings.



(a) B.1: Two Large Rings, Constructs.

(b) B.1: Two Large Rings, Correct Pointers.

(c) B.1: Two Large Rings, Messages.

(d) B.1: Two Large Rings, Bandwidth Consumption.

Figure 5.6: B.1: Comparison of Ring Reunion Algorithm and Ring Unification Algorithm.

5.2.4 B.1 Comparison of Performance

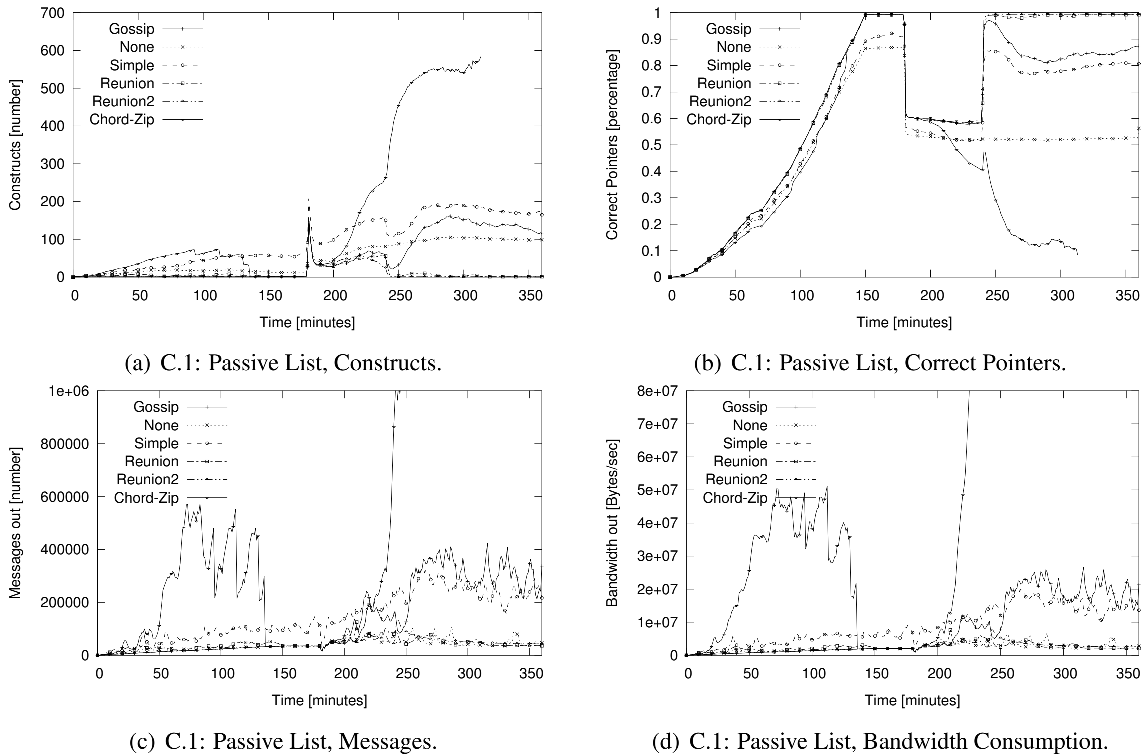
We decided to compare the gossip-based Ring Unification Algorithm to the Ring Reunion Algorithm with different parallelization parameters in a scenario with 10424 nodes to get better insights on the performance of both algorithms. As one can obtain from Figures 5.6(a) and 5.6(b) the Ring Reunion Algorithm is the faster, the more instances it starts. Nevertheless, Figure 5.6(c) shows that not more messages are sent with a high number of merger instances, compared to a small number of instances.

5.2.5 C.1: Study of Passive List

Below, we will have a closer look onto scenarios in which different methods are used to find alive nodes automatically after a network partitioning event. In Figure 5.7 the usage of a passive list is shown in combination with the Chord-Zip Algorithm, the simple Ring Unification Algorithm, the gossip-based Ring Unification Algorithm, the Ring Reunion Algorithm with one instance and four parallel instances. Figure 5.7(a) reveals that the Chord-Zip Algorithm is not suitable for being combined with the passive list, as it is to slow to react on multiple, simultaneous opportunities to start different merger instances. Surprisingly, both Ring Unification Algorithms have not been able to handle the network isolation event within the simulation time. In Figure 5.7(b) it can be seen that during the isolation event about 40 percent of all correct pointers have been rearranged badly. After 240 minutes all network partitions have become reachable again, as the results of the Ring Reunion Algorithm indicate, which is capable of reordering the successor pointers correctly. Furthermore one can see that the Ring Unification Algorithm only adjusts approximately 20 percent of the failed successor pointers. However, the Chord-Zip Algorithm is not capable of adjusting the broken connections, as to many instances are started which disrupt the order of the successor pointers again. The amount of messages which are sent throughout the simulation is presented in Figure 5.7(c). During the whole simulation the amount of messages, sent by the Chord-Zip Algorithm, rises quickly, whereas the Ring Reunion Algorithm does not produce much more messages than Chord by itself. After the 240th minute, when all partitions are connected again, the Ring Unification Algorithms produce more messages than usual, because of their efforts to unify the created constructs.

5.2.6 C.2: Study of Active Contact List

As an alternative to the passive list we tested an *active contact list* of 160 randomly chosen nodes which are obtained by each node from the bootstrap node during the join phase. This list is iterated periodically in Setup C.2, in order to obtain suitable contact nodes. As Figure 5.8(b) indicates, only the Ring Reunion Algorithm manages to adjust all successor pointers within the simulated time. Again the gossip-based Ring Unification Algorithm forms a large amount of constructs after all network partitions have become reachable again, see Figure 5.8(a). It might be possible that the gossip-based Ring Unification Algorithm is capable of merging the separated overlays again, but that would take a long time. Further, Figures 5.8(a), 5.8(c) and 5.8(d) show that the message and bandwidth consumption of both algorithms highly relate to the number of constructs in the underlying network.



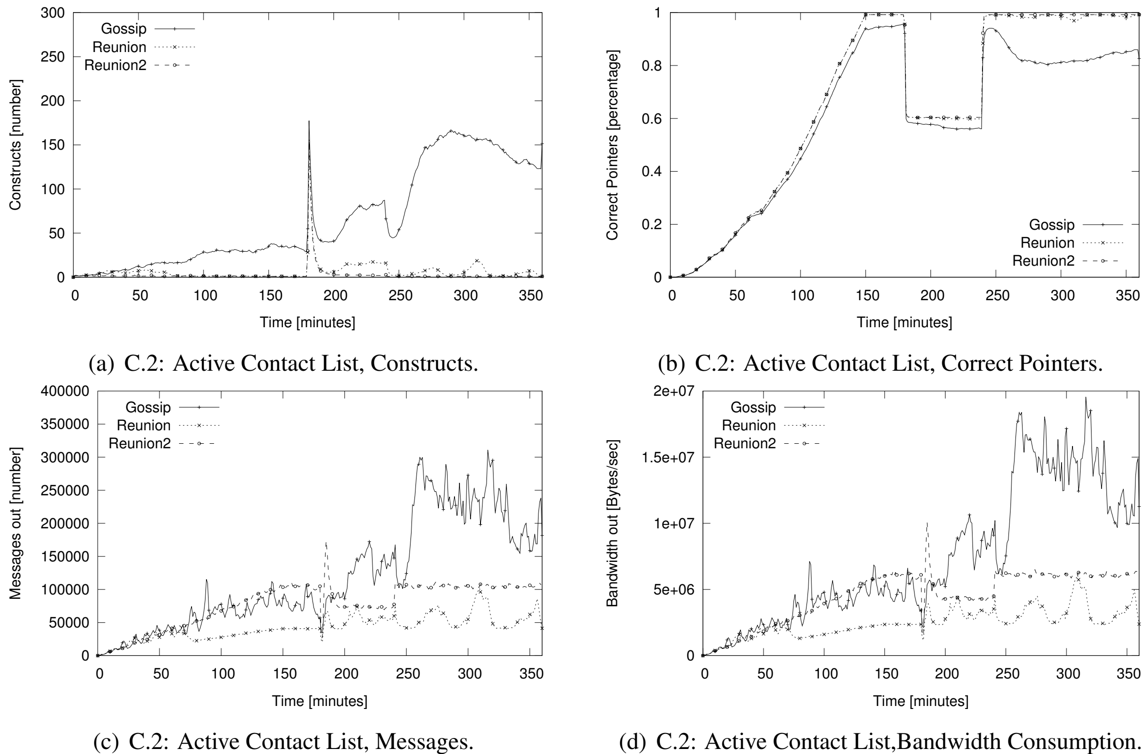
(a) C.1: Passive List, Constructs.

(b) C.1: Passive List, Correct Pointers.

(c) C.1: Passive List, Messages.

(d) C.1: Passive List, Bandwidth Consumption.

Figure 5.7: C.1: Passive list used to find alive contact nodes.



(a) C.2: Active Contact List, Constructs.

(b) C.2: Active Contact List, Correct Pointers.

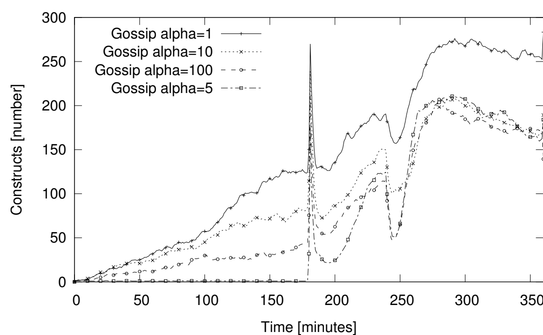
(c) C.2: Active Contact List, Messages.

(d) C.2: Active Contact List,Bandwidth Consumption.

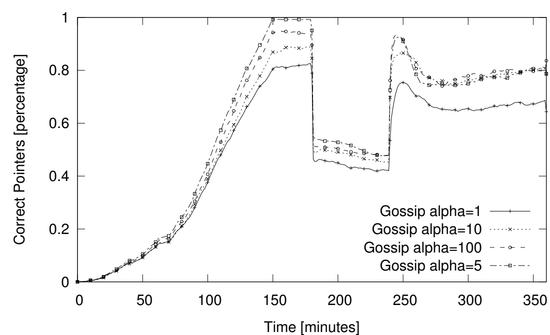
Figure 5.8: C.2: Active contact list is iterated periodically to find further nodes. No probability is used to reduce message overhead.

5.2.7 C.3-5: Study of Probability

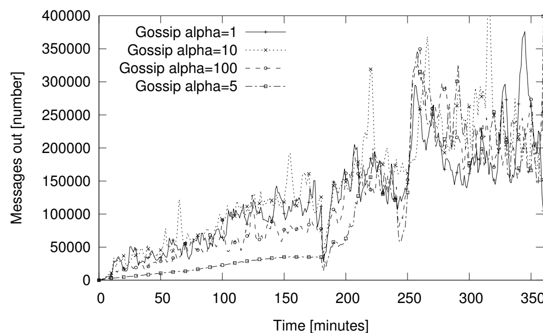
Next, in Setups C.3-5, we reduce the amount of messages by reducing the number of instances that are started by a specific algorithm. Therefore we extended the algorithm with the ability to estimate the size of the current construct a node is in. Now, each node picks a random number out of $[0,1]$ and starts a merger instance only if the chosen random number is less than $\alpha / size_c$, where α constitutes the number of started mergers per overlay construct and $size_c$ is the estimated number of nodes in the current construct. Our goal of the simulations which are represented in Figure 5.9 was to determine a good value for α, i.e. the number of merger instances per construct. As can be seen in Figure 5.9(a) the Ring Unification Algorithm performs the poorer the less merger instances are started. Considering Figure 5.11(b) we see that the Ring Reunion Algorithm performs well if approximately 10 instances per construct are started in combination with 4 parallel instances. Further we learn from this evaluation result that for a merging algorithm it is necessary to react as quickly as possible on partitioning events to perform well and to reduce costs.



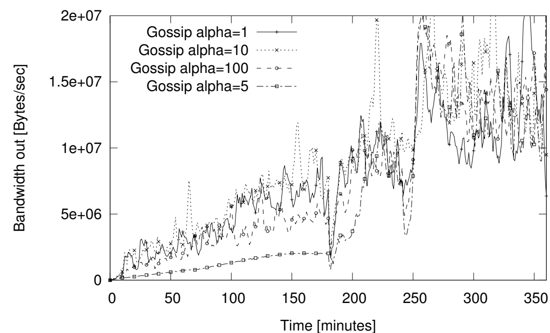
(a) C.3: Gossip-based Ring Unification, Constructs.



(b) C.3: Gossip-based Ring Unification, Correct Pointers.

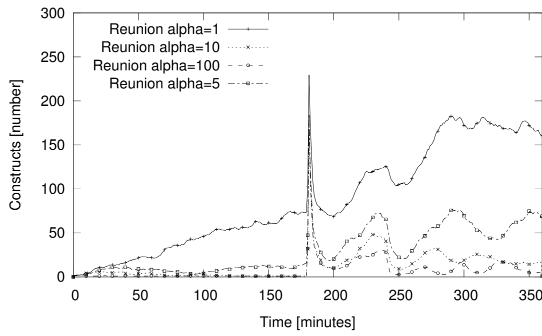


(c) C.3: Gossip-based Ring Unification, Messages.

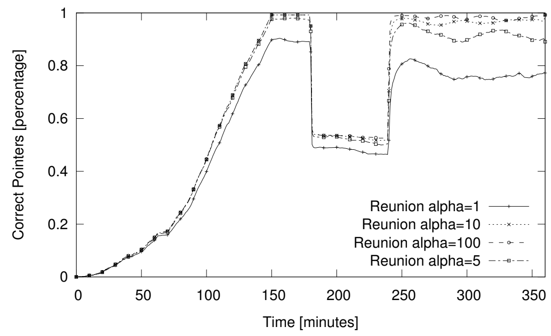


(d) C.3: Gossip-based Ring Unification, Bandwidth Consumption.

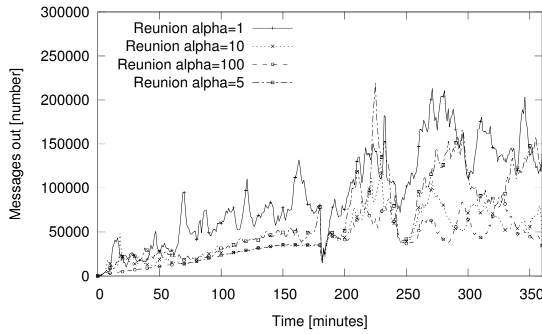
Figure 5.9: C.3: Gossip-based Ring Unification Algorithm, overlay constructs start α merger instances every 4 minutes on average.



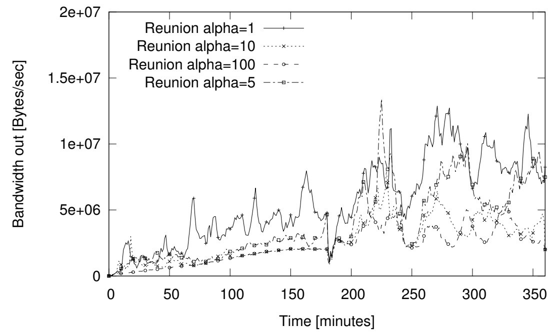
(a) C.4: Ring Reunion, Constructs.



(b) C.4: Ring Reunion, Correct Pointers.

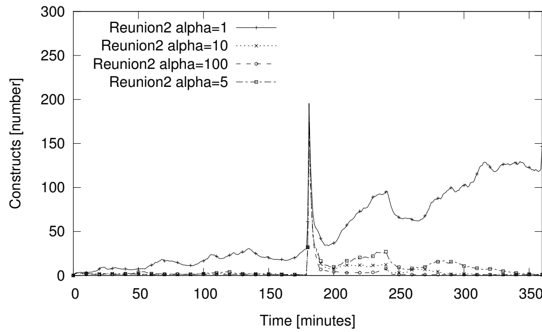


(c) C.4: Ring Reunion, Messages.

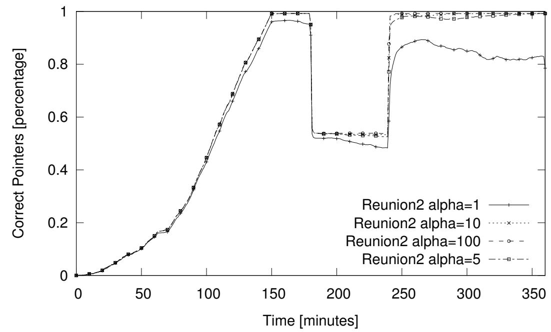


(d) C.4: Ring Reunion, Bandwidth Consumption.

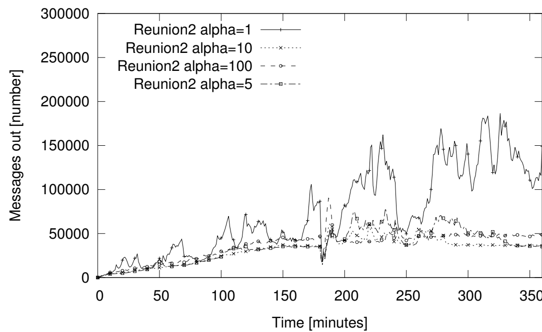
Figure 5.10: C.4: Simple Ring Reunion Algorithm in combination with alive contact list.



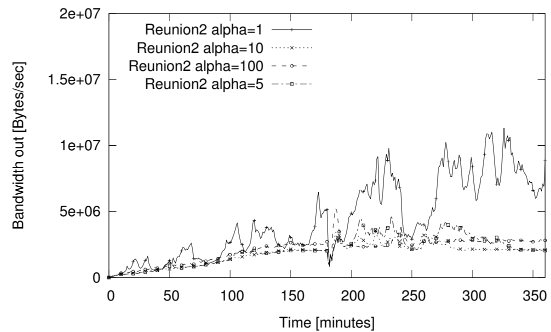
(a) C.5: Ring Reunion2, Constructs.



(b) C.5: Ring Reunion2, Correct Pointers.



(c) C.5: Ring Reunion2, Messages.

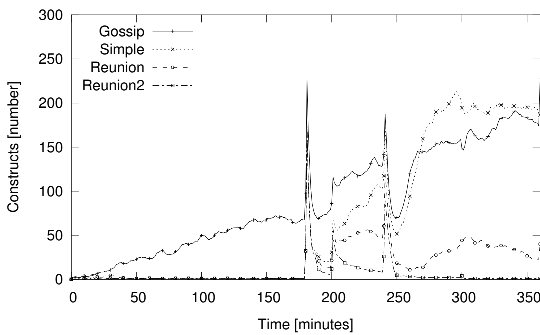


(d) C.5: Ring Reunion2, Bandwidth Consumption.

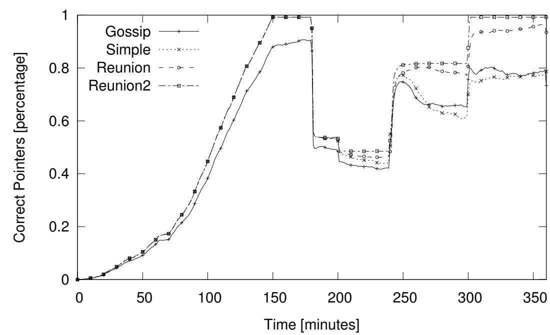
Figure 5.11: C.5: Ring Reunion with 4 parallel instances in combination with alive contact list.

5.2.8 D.1: Complex and Realistic Scenario

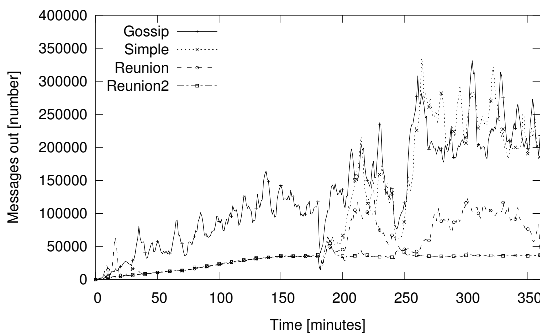
The last Setup D.1 considers a complex scenario in which multiple regions become separated due to network isolations during the merging period. Figures 5.12(a) and 5.12(b) verify our expectations. The Ring Reunion Algorithm with 4 parallel instances is able to merge all reachable regions fast enough to handle even multiple network failures. Figure 5.12(a) also shows that the Ring Reunion Algorithm reduces the number of constructs quickly after it rises at minutes 180, 200 and 240. In addition one can obtain from Figure 5.12(c) that the Ring Reunion Algorithm, if configured properly, does not produce much more messages during network partitioning events than usual. In conclusion, our evaluation shows that the Ring Reunion Algorithm is fast enough to handle even complex use cases with low traffic overhead.



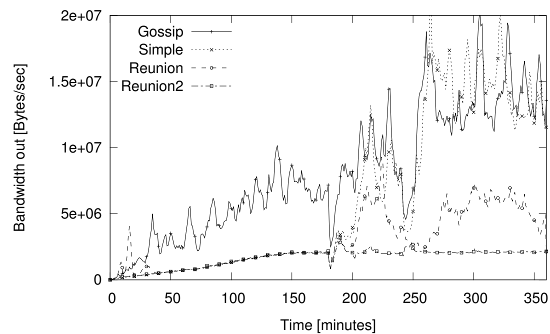
(a) D.1: Complex Isolation, Constructs.



(b) D.1: Complex Isolation, Correct Pointers.



(c) D.1: Complex Isolation, Messages.



(d) D.1: Complex Isolation, Bandwidth Consumption.

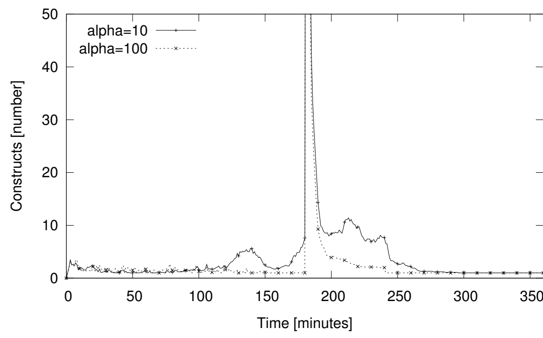
Figure 5.12: D.1: Complex realistic scenario with usage of alive contact list and 10 started merger instances per construct.

5.2.9 E.1-2: Parameter Studies and Churn

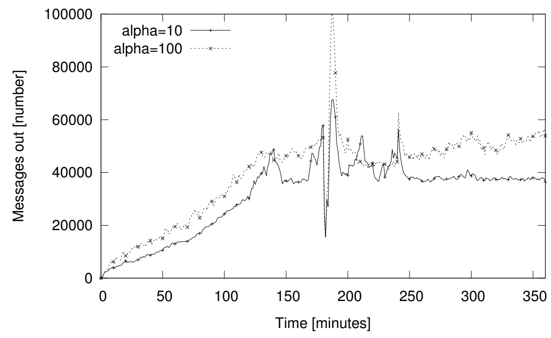
In this section we examine the behavior of the Ring Reunion Algorithm in the context of churn and different parameter settings, in order to determine a suitable configuration of our algorithm for realistic use cases. Figure 5.13 reveals the behavior of the Ring Reunion Algorithm if multiple parallel instances are distributed by the first initiator node. In this scenario, every 5 minutes the *active contact list* tests a contact node to be reachable. In Figures 5.13(a) and 5.13(c) one can see that the merger algorithm is able to unify the disrupted network, no matter if 10 instances per construct are started or 100. On the contrary, if the number of parallel instances is too high, the merger algorithm operates poorly in some cases. In 50 percent of our simulations with 64 parallel instances, the number of constructs suddenly rises after the network isolation stops, so that in the end the merger shows to be unsuccessful, as to be seen in Figure 5.13(g). Nevertheless, Figures 5.13(b), 5.13(d), 5.13(f) and 5.13(h) prove that the message overhead produced by the Ring Reunion Algorithm depends on the number of overlay constructs and can be limited by reducing the number of merger attempts per construct by adjusting parameter α. Figure 5.14 shows the results of a simulation, in which multiple parallel instances have been tested in combination with different intervals for starting merger instances with $\alpha = 10$. Considering the number of constructs in Figure 5.14, it can be observed that high numbers of parallel instances operate the better, the less attempts are started to merge the overlay. Considering the quantity of messages on the other hand reveals that the number of simultaneous instances does not affect traffic overhead and the resulting bandwidth consumption.

The behavior of the Ring Reunion Algorithm can be explained as follows: if the network is not yet fully stabilized after a partitioning event, and in addition multiple merger instances are started, the current overlay constructs are suddenly reordered. As a consequence the number of overlay constructs rises in this short period of time, as can be seen best in 5.14(g) for all intervals greater than 5 minutes. Furthermore, in some cases, the determination of the additional merger instances takes longer time than the actual merger process or the interval within which new merger attempts are started. Hence, the additional started merger instances tear up the current overlay constructs again. In few cases this behavior leads to a dysfunction of the merger process which is caused by wrong parameter choices. Another reason for this wrong behavior are changes in the routing table, due to churn. If one node leaves the overlay suddenly, it might happen that too many instances try to unify the falsely detected overlay partition again.

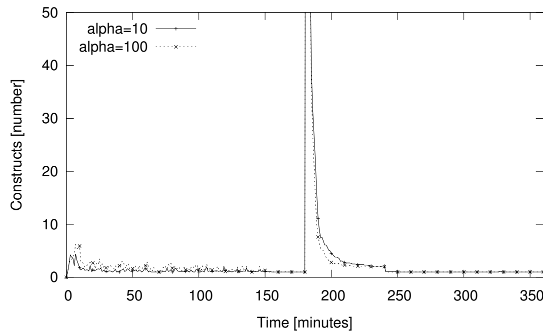
To conclude our studies, we suggest to use the Ring Reunion Algorithm in combination with an *active contact list*, which attempts to start approximately 10 new merger instances every 4 to 5 minutes per overlay construct. The parameter to limit the number of additional merger instances should be set to 3 or 4, so that not more than 8 to 16 instances are created simultaneously and the Ring Reunion Algorithm is able to operate smoothly, that is not producing to much overlay constructs, fast and reliably.



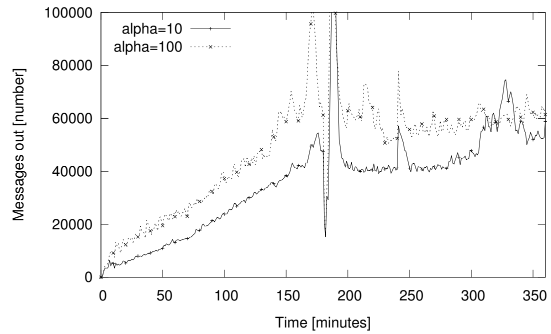
(a) E.1: 8 Instances, Constructs.



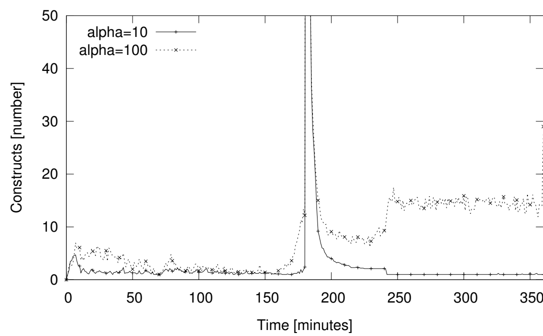
(b) E.1: 8 Instances, Messages.



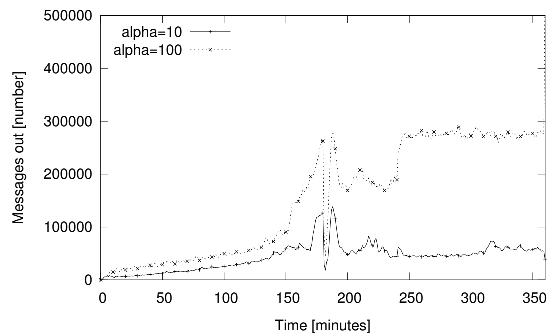
(c) E.1: 16 Instances, Constructs.



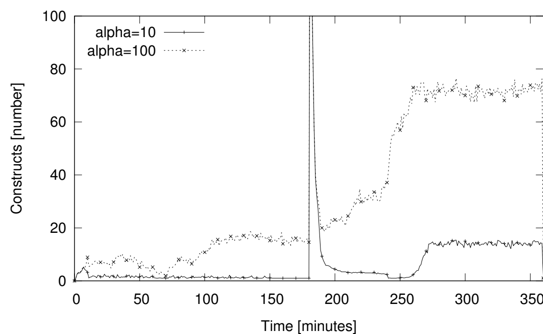
(d) E.1: 16 Instances, Messages.



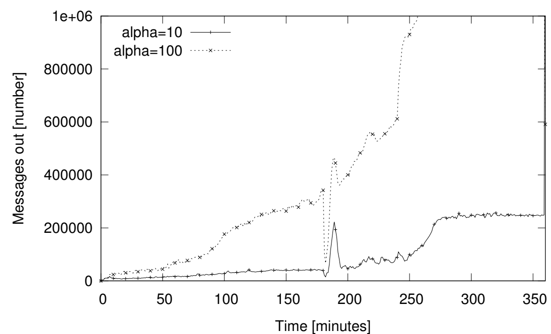
(e) E.1: 32 Instances, Constructs.



(f) E.1: 32 Instances, Messages.

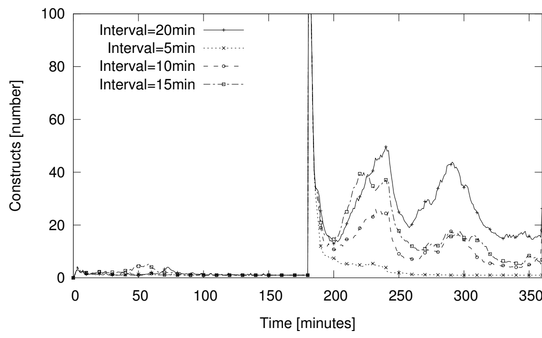


(g) E.1: 64 Instances, Constructs.

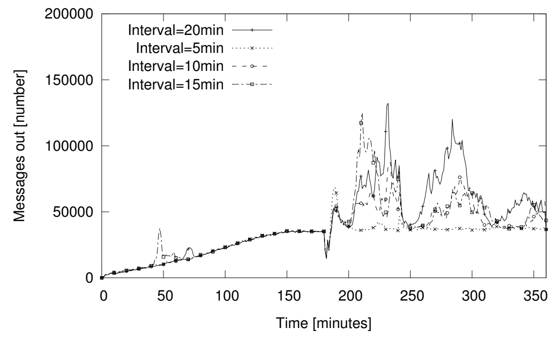


(h) E.1: 64 Instances, Messages.

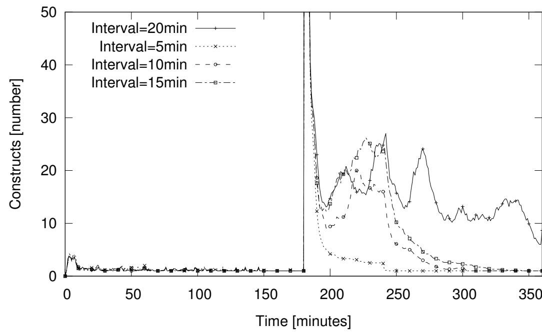
Figure 5.13: E.1: Ring Reunion Algorithm with parallel instances, $\alpha \in \{10, 100\}$.



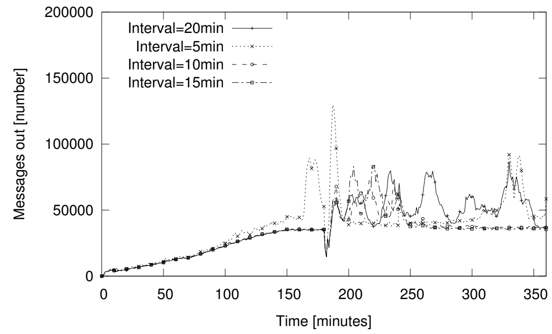
(a) E.2: 8 Instances, Constructs.



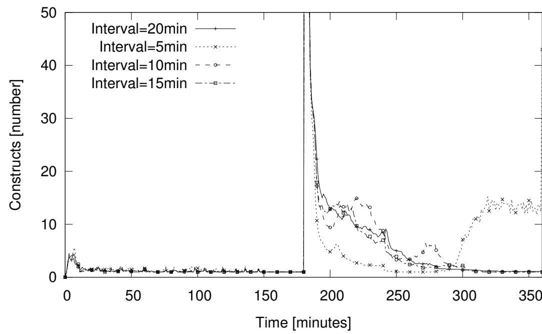
(b) E.2: 8 Instances, Messages.



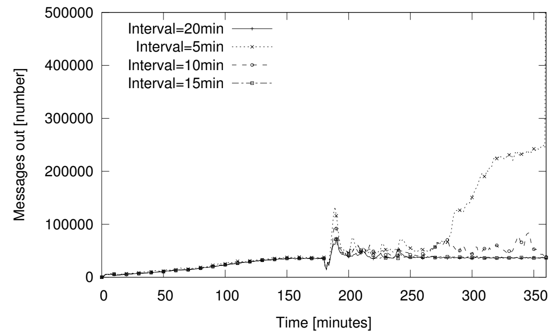
(c) E.2: 16 Instances, Constructs.



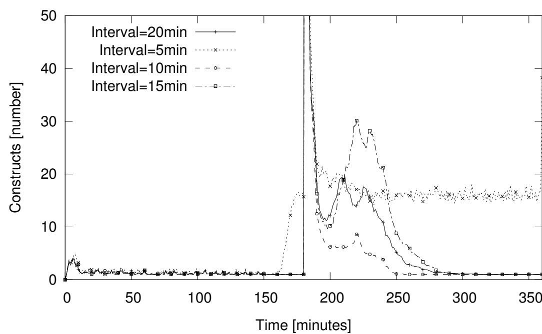
(d) E.2: 16 Instances, Messages.



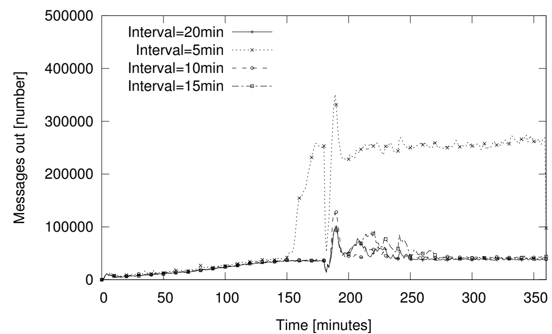
(e) E.2: 32 Instances, Constructs.



(f) E.2: 32 Instances, Messages.



(g) E.2: 64 Instances, Constructs.



(h) E.2: 64 Instances, Messages.

Figure 5.14: E.2: Ring Reunion Algorithm with parallel instances and various values for interval parameter, $\alpha = 10$.

Chapter Conclusion

We have investigated the quality of Chord-Zip, the Ring Unification Algorithm and the Ring Reunion Algorithm by simulating different scenarios in this chapter. The setups of our simulations have been described in Section 5.1. In Section 5.1.1 we have described scenarios in which the merging process is manually initiated in order to measure the operational time of each merging algorithm. We have enabled automatic partition detection and merging in Section 5.1.2, to simulate more realistic cases.

The results of our simulations have been discussed in Section 5.2. In Sections 5.2.1, 5.2.2, 5.2.3 and 5.2.4 we have investigated Chord-Zip, the Ring Unification and the Ring Reunion Algorithm on networks with 2, 3 and 5 partitions and manually started merging procedures. Our results show that only the gossip-based Ring Unification Algorithm and our Ring Reunion Algorithm are capable of merging multiple networks if contact nodes are given to suitable initiator nodes manually by a supervising person, e.g an administrator. In Sections 5.2.5, 5.2.6, 5.2.7 and in Section 5.2.8 we have demonstrated that our Ring Reunion Algorithm is the only merger which is able to unify multiple network partitions, even in complex cases.

We have proved in Section 5.2.9 that our algorithm performs well in the presence of churn. In addition, we have suggested a good configuration for the Ring Reunion Algorithm: it should be combined with an *active contact list*, which attempts to start approximately 10 new merger instances per overlay construct every 4 to 5 minutes. The parameter to limit the number of additional merger instances should be set to 3 or 4 to operate smoothly, fast and reliably.

Chapter 6

Conclusion

In this thesis we are concerned with the question, which criteria have to be met, to unify overlay networks, especially ring-based overlays like Chord, which have been separated due to network partitioning. To answer this question we compare existing studies on this topic. On the one hand the authors of [23] claim the function of Chord overlays would be disrupted during a network merger. On the other hand two solutions are given in [11] and [10] by introducing merging algorithms that are promised to be able to unify multiple Chord rings quickly. Nevertheless none of those studies examine any characteristics of network partitions or criteria which foster a merger algorithm's performance. As a result we present our Ring Reunion Algorithm, a novel merging algorithm. We have implemented it in an event-based peer-to-peer simulator and evaluate it extensively in this work. Furthermore we compare our approach to the existing algorithms by examining simulations of realistic scenarios.

We show in our study that whenever multiple geographical regions are isolated for a long time, it might occur that outdated finger entries are replaced by newer ones, which are only present in the own region. If in addition the separated regions are connected again they do not converge to a common ring, since all contact data is obtained only from the region one node is in. A merging algorithm is needed to unify the overlays again. Further, it is possible that within a separated region, multiple independent overlays are established. This behavior can be explained simply: in a Chord ring all nodes are equally distributed, so that most of them maintain more routing information about nodes from the other region than about nodes from the own geographic region. Hence, during isolation events only those nodes are formed to a group, which are represented in the same set of finger entries and successor or predecessor pointers.

Our newly designed merging algorithm is able to merge such groups of overlays effortlessly. Other separated overlays are found by iterating a small list of randomly chosen contact nodes, which could be obtained during the join phase or periodically updated via a background-service. Another option would be to use a passive list as described in [11] or even a combination of both approaches. Further, our algorithm estimates the size of the network one node is currently in, in order to determine how

often a merging algorithm should be started to reduce message overhead. To increase performance of our Ring Reunion Algorithm we have extended it with the feature to inform other nodes in a network to start further merger instances simultaneously. Whenever a node tries to merge another node within the same overlay, the merging algorithm is quickly terminated to reduce further costs.

As a last step the Ring Reunion Algorithm is evaluated and compared with the Ring Unification Algorithm [11] and the Chord-Zip Algorithm [10]. The scenarios we simulate are divided into two major groups. First, we evaluate the fundamental behavior of each algorithm by merging two, three and five separated Chord rings manually. Second we investigate the algorithms in more realistic scenarios in which one or multiple groups are suddenly isolated and alive nodes are found via passive list or *active contact list*. Unfortunately the Chord-Zip Algorithm is inaccurately described in [10] so that in our simulations it occurs to be too slow to react on multiple network partitions and does not manage to terminate started merger instances properly. Although the Ring Unification Algorithm is well described and suitable for the merging of multiple rings in a controlled environment, it suffers from unifying multiple constructs during realistic scenarios in which arbitrary parts of a network become isolated. Nevertheless, the Ring Reunion Algorithm is able to do so, even in complex cases and with the presence of churn. As the evaluation proves, our mechanism to reduce message overhead works fine in combination with multiple parallel instances.

Future Work

The overlay merging algorithm, which has been presented in this thesis, has been simulated and evaluated with Chord mainly. Nevertheless we have developed the algorithm with respect to other ring-based overlays, so that its techniques are transferable to them. It would be interesting to use the Ring Reunion Algorithm as a basis to create an application for ring-based overlays like Pastry, Chord or any of Chord's modifications, which manages to merge the respective overlay.

We focus in this work on the routing ability of ring-based overlays. Another issue which needs to be investigated, with respect to network partitioning in structured overlays, is the preservation of data availability during a merger and the synchronization of data items afterwards. At this point we can not give any advice or solution on how data consistency can be protected in case of network partitioning events. Obviously, this problem depends on the application which handles the respective data items. It would be possible to traverse a list of successor nodes to find a requested data item. Since current overlay designs rarely consider network partitioning, it is still necessary to investigate which criteria have to be met to assure data consistency and availability during and after a merging process.

Bibliography

[1] Amnesty International, "Amnesty International Report 2013." https://www.amnesty.org/en/annual-report/2013/downloads, 2013.

[2] X. Xu, Z. M. Mao, and J. A. Halderman, "Internet Censorship in China: Where Does the Filtering Occur?," in *Passive and Active Measurement*, Springer, 2011.

[3] R. Clayton, S. J. Murdoch, and R. N. Watson, "Ignoring the Great Firewall of China," in *Privacy Enhancing Technologies*, Springer, 2006.

[4] P. N. Howard, A. Duffy, D. Freelon, M. Hussain, W. Mari, and M. Mazaid, "Opening Closed Regimes: What Was the Role of Social Media During the Arab Spring?," *PITPI*, 2011.

[5] E. Stepanova, "The Role of Information Communication Technologies In the 'Arab Spring'," *Implications beyond the Region. Washington, DC: George Washington University, PONARS Eurasia Policy Memo*, no. 159, pp. 1–6, 2011.

[6] M. Richtel, "Egypt Cuts Off Most Internet and Cell Service." http://www.nytimes.com/2011/01/29/technology/internet/29cutoff.html, 2011.

[7] CircleID, "Egyptian Government Shuts Down Most Internet and Cell Services." http://www.circleid.com/posts/egyptian_government_shuts_down_most_internet_and_cell_services/, 2011.

[8] J. Glanz and J. Markoff, "Egypt Leaders Found 'Off' Switch for Internet." http://www.nytimes.com/2011/02/16/technology/16internet.html, 2011.

[9] BBC News, "Asia Communications Hit by Quake." http://news.bbc.co.uk/2/hi/asia-pacific/6211451.stm, 2006.

[10] Z. Kis and R. Szabo, "Chord-Zip: A Chord-Ring Merger Algorithm," *Communications Letters, IEEE*, vol. 12, no. 8, pp. 605–607, 2008.

[11] T. M. Shafaat, A. Ghodsi, and S. Haridi, "Dealing With Network Partitions in Structured Overlay Networks," *Peer-to-peer networking and applications*, vol. 2, no. 4, pp. 334–347, 2009.

[12] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *ACM SIGCOMM '01: Proc. of the Int. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ACM, 2001.

[13] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," in *IFIP/ACM Middleware '01: Proc. of the Int. Conf. on Distributed Systems Platforms*, vol. 2218 of *LNCS*, Springer, 2001.

[14] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," in *IPTPS '02: Proc. of the Int. Workshop on Peer-To-Peer Systems*, vol. 2429 of *LNCS*, Springer, 2002.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A Scalable Content-Addressable Network," in *ACM SIGCOMM '01: Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, vol. 31, ACM, 2001.

[16] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt, "P-Grid: A Self-Organizing Structured P2P System," *SIGMOD Record*, vol. 32, no. 3, 2003.

[17] J. Wang and Z. Yu, "A New Variation of Chord With Novel Improvement on Lookup Locality," in *GCA '06: Proc. of the Int. Conf. on Grid Computing & Applications*, CSREA Press, 2006.

[18] B. Leong, B. Liskov, and E. D. Demaine, "Epichord: Parallelizing the Chord Lookup Algorithm With Reactive Routing State Management," *Computer Communications*, vol. 29, no. 9, pp. 1243–1259, 2006.

[19] S. Kniesburges, A. Koutsopoulos, and C. Scheideler, "Re-Chord: A Self-Stabilizing Chord Overlay Network," in *ACM SPAA '11: In Proc. of the Symposium on Parallelism in Algorithms and Architectures*, ACM, 2011.

[20] M. Onus, A. W. Richa, and C. Scheideler, "Linearization: Locally Self-Stabilizing Sorting in Graphs," in *SIAM ALENEX '07: Proc. of the Workshop on Algorithm Engineering and Experiments*, vol. 7, SIAM, 2007.

[21] M. Benter, M. Divband, S. Kniesburges, A. Koutsopoulos, and K. Graffi, "Ca-Re-Chord: A

Churn Resistant Self-Stabilizing Chord Overlay Network," in *NetSys '13: Proc. of the Int. Conf. on Networked Systems*, IEEE, 2013.

[22] V. A. Mesaros, B. Carton, and P. Van Roy, "S-Chord: Using Symmetry to Improve Lookup Efficiency in Chord," in *PDPTA '03: Proc. of the Int. Conf. Parallel and Distributed Processing Techniques and Applications*, CSREA Press, 2002.

[23] A. Datta and K. Aberer, "The Challenges of Merging Two Similar Structured Overlays: A Tale of Two Networks," in *IEEE SOS '06, Proc. of the Int. Conf. on Self-Organizing Systems*, IEEE, 2006.

[24] A. Binzenhöfer, D. Staehle, and R. Henjes, "Estimating The Size of A Chord Ring," *University of Würzburg, Tech. Rep*, 2004.

[25] UC Berkeley, LBL, USC/ISI, and Xerox PARC, "The Network Simulator - ns-2." http://www.isi.edu/nsnam/ns/, 2014.

[26] NS-3 Consortium, "The Network Simulator - ns-3." http://www.nsnam.org/, 2014.

[27] S. K. Ingmar Baumgart, Bernhard Heep, "OverSim: A Scalable and Flexible Overlay Framework for Simulation and Real Network Applications," in *IEEE P2P '09: Proc. of the 9th Int. Conference on Peer-to-Peer Computing*, IEEE, 2009.

[28] A. Varga, "The OMNeT++ Discrete Event Simulation System," *ESM '01: Proceedings of the European Simulation Multiconference*, 2001.

[29] K. Graffi, "PeerfactSim.KOM: A P2P System Simulator – Experiences and Lessons Learned," in *IEEE P2P '11: Proc. of the Int. Conf. on Peer-to-Peer Computing*, IEEE, 2011.

[30] Clip2 - The Gnutella Developer Forum, "Gnutella." http://rfc-gnutella.sourceforge.net/developer/stable/index.html, 2002.

[31] T. E. Ng and H. Zhang, "Predicting Internet Network Distance With Coordinates-Based Approaches," in *IEEE INFOCOM '02: Proc. of the Int. Conf. on the Joint Conference of the IEEE Computer and Communications Societies*, IEEE, 2002.

[32] W. Matthews and L. Cottrell, "The PingER Project: Active Internet Performance Monitoring For the HENP Community," *IEEE: Communications Magazine*, vol. 38, no. 5, pp. 130–136, 2000.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 4.March 2014 Tobias Amft

Please add here

the DVD holding sheet

This DVD contains:

- A *PDF* version of this master thesis
- All LaTeXand grafic files that have been used, as well as the corresponding scripts
- The source code of the software that was created during the master thesis
- The simulation data that was created during the evaluation
- The referenced websites and papers