ROYAL INSTITUTE
OF TECHNOLOGY

Partition Tolerance and Data Consistency in Structured Overlay Networks

TALLAT MAHMOOD SHAFAAT

Doctoral Thesis in
Electronic and Computer Systems
KTH – Royal Institute of Technology
Stockholm, Sweden, June 2013

TRITA-ICT/ECS AVH 13:09
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-13/09-SE
ISBN 978-91-7501-725-9

KTH School of Information and
Communication Technology
SE-100 44 Stockholm
Sweden

SICS Dissertation Series 63
ISSN 1101-1335
ISRN SICS-D–63–SE

Swedish Institute of Computer Science
SE-164 29 Kista
Sweden

Abstract

Structured overlay networks form a major class of peer-to-peer systems, which are used to build scalable, fault-tolerant and self-managing distributed applications. This thesis presents algorithms for structured overlay networks, on the routing and data level, in the presence of network and node dynamism.

On the routing level, we provide algorithms for maintaining the structure of the overlay, and handling extreme churn scenarios such as bootstrapping, and network partitions and mergers. Since any long lived Internet-scale distributed system is destined to face network partitions, we believe structured overlays should intrinsically be able to handle partitions and mergers. In this thesis, we discuss mechanisms for detecting a network partition and merger, and provide algorithms for merging multiple ring-based overlays. Next, we present a decentralized algorithm for estimating the number of nodes in a peer-to-peer system. Lastly, we discuss the causes of routing anomalies (lookup inconsistencies), their effect on data consistency, and mechanisms on the routing level to reduce data inconsistency.

On the data level, we provide algorithms for achieving strong consistency and partition tolerance in structured overlays. Based on our solutions on the routing and data level, we build a distributed key-value store for dynamic partially synchronous networks, which is linearizable, self-managing, elastic, and exhibits unlimited linear scalability. Finally, we present a replication scheme for structured overlays that is less sensitive to churn than existing schemes, and allows different replication degrees for different key ranges that enables using higher number of replicas for hotspots and critical data.

Keywords: structured overlay networks, distributed hash tables, network partitions and mergers, size estimation, lookup inconsistencies, distributed key-value stores, linearizability, dynamic reconfiguration, replication.

Acknowledgements

I am highly indebted to Professor Seif Haridi for giving me the opportunity to work under his supervision. His expanse of knowledge and methodology of supervision is remarkable. Not only did I learn a lot from him, I also tremendously enjoyed my time as a student. I am also extremely grateful to Ali Ghodsi for providing me enormous help and encouragement during this thesis. His intellect, enthusiasm and approach to solving problems has been, and will always be, a source of inspiration for me. I would like to thank Prof. Vladmimir Vlassov as well, for providing valuable feedback in general and this thesis in particular. I would also like to show my gratitude to Sverker Janson for offering me the chance to be a member of CSL at SICS.

During my time as a graduate student, I had the pleasure to visit industrial labs to get a different perspective on real-world problems and research environments. I feel extremely privileged to have worked with Ganesh Venkitachalam (VMware, Inc., Palo Alto, 2010), Alex Mirgorodskiy (VMware, Inc., Palo Alto, 2011), Phil Bernstein and Sudipto Das (Microsoft Research, Redmond, 2012), and Sergey Bykov (Microsoft Research, Redmond, 2013). I have learnt a lot from them. Their insights during our discussions have greatly influenced my way of reasoning.

I would also like to thank my colleagues, both at KTH and SICS, for a lot of fruitful and conducive discussions; Ahmad Al-Shishtawy, Cosmin Arad, Amir Payberah, Fatemeh Rahimian, Daniela Bordencea, Jim Dowling, Niklas Ekström, Sarunas Girdzijauskas, and Joel Höglund. I had a great experience collaborating with Cosmin towards the end of my thesis. Special thanks to my friends in Sweden, for making my stay in Sweden cherishable for the rest of my life: Kashif, Waseem, Tahir, Umair, Magnus, Hedvig, Rick, Anton, Simon, Sarah, Chris, Peggy, Jeff, Britta, Francious, Maria, Haseeb, and Salman.

Finally, I would like to dedicate this work to my parents, my sisters and brother. Their continuous support and belief in me has been a tremendous source of inspiration.

To my parents

Contents

CHAPTER 1

Introduction

With the advent of the Internet, applications provide services to remote client machines over the network. These applications build a distributed system where one or more computers, also known as *nodes*, provide some service to other computers over the Internet. This service paradigm presents great challenges. One such challenge is to build *scalable* systems such that the service quality of an application does not degrade as the number of clients using the service increases. Furthermore, as the Internet is spread geographically, and it uses various network components being managed by independent administrators, failure of nodes and network links is a norm in such systems. Thus, achieving *fault-tolerance* is vital.

One of the first approaches to built distributed systems was a client-server paradigm. While the client-server paradigm is still popular and effective, it has its drawbacks. The main disadvantage of server-based systems is dependence on one (or a few) server(s) to provide a service to a large number of clients. Using a single server leads to a single point of failure and is not scalable. Furthermore, the machines used as servers have to be tremendously powerful in terms of network connectivity, storage capacity, and processing power to handle growing number of clients, and their data. Thus, using high end servers is an expensive approach. This lead to finding alternate paradigms, one of them being *peer-to-peer systems*.

This thesis focuses on achieving fault-tolerance in peer-to-peer systems. In this chapter, we give a brief introduction to the peer-to-peer approach for building large-scale distributed systems. Although this

approach can be used on any network infrastructure over which entities/nodes can communicate, such as an adhoc wireless system, we use the Internet as a reference in our discussions. After providing an overview of various peer-to-peer systems, we present the research objectives of this thesis work, and our contributions to meet the research objectives. Thereafter, we discuss the outline of the thesis.

1.1 Peer-to-peer Systems

With the advancement of technology, network connectivity, storage, and processing power have become cheaper. As a result, computers at the edge of the network, e.g. personal computers, are more powerful. This has lead to the vision of using resources available at the edge of the network, resulting in the realization of systems known as *peer-to-peer systems*. Peer-to-peer (P2P) systems are decentralized and a node may act as both, a server and a client. Thus, a node can use services provided by other nodes, while it also provides services to other nodes.

Since many of the edge machines are less reliable compared to dedicated servers, achieving fault-tolerance becomes a non-trivial challenge in peer-to-peer systems. Furthermore, since there is no single point of control, edge machines can join and leave the system as they please. Thus, another crucial challenge is that the system should provide easy management, with machines coming up and going down at any time. This lead to the development of another attractive feature of P2P systems, namely *self-management*, where the system requires minimum manual configuration and management.

One of the first peer-to-peer systems, called Napster [113], appeared in 1999. Napster was mainly used for sharing music files. While Napster removed the burden of hosting the shared files on the servers, it still used dedicated servers for the indexing service. The next challenge was to make a decentralized, scalable, and fault-tolerant indexing service. This would enable a node to publish information about a data item, e.g. file, in a decentralized fashion. Similarly, a node would be able to find/lookup information about an item published earlier in the system. To achieve this, nodes that are part of the system are connected to each other over the Internet instead of connecting to the server(s). Thus, nodes have network connection information about some other nodes, called *neighbours* of the node, participating in the system. The information about neighbours of a node are stored locally in a data-structure called the *routing table* of the node. The routing table includes names and network connection information about neighbours, thus enabling a

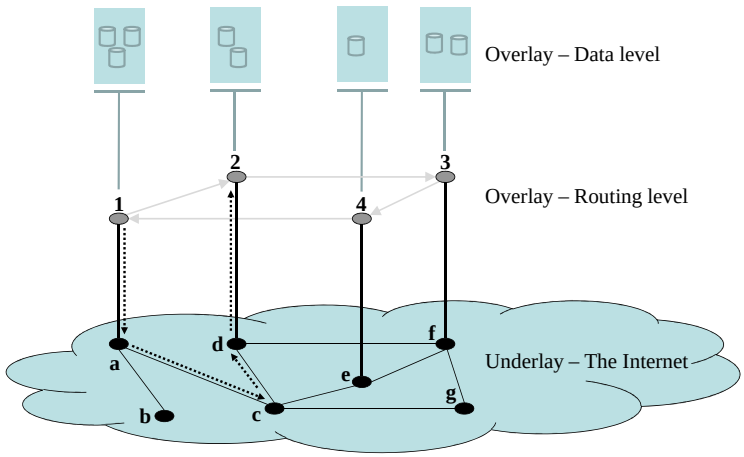
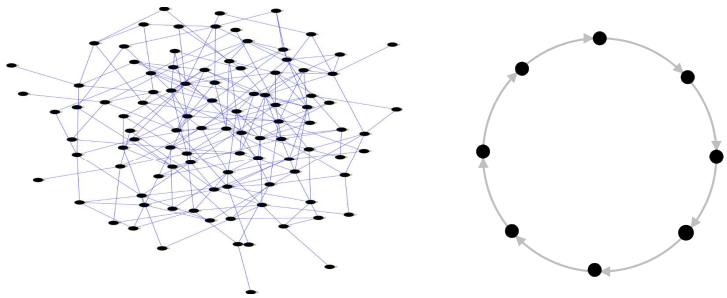


Figure 1.1: An overlay network built on top of an underlay network. The overlay consists of nodes 1, 2, 3 and 4, while the underlay consists of components (nodes/routers/switches etc.) a, b, c, d, e, f and g. The overlay consists of a routing level and data level. The routing level is used for sending messages between nodes, e.g., 1 can send a message to a neighbour 2. Such a message travels through the underlay components a, c, and d. The data level is concerned with storing items in the overlay nodes.

node to route messages to other nodes. In essence, the routing tables of all nodes create a routing system on top of the existing network infrastructure, e.g. the Internet. The overall network routing view created by routing tables of all the nodes is known as an *overlay* network. Since an overlay uses the Internet to route messages, the Internet is referred to the *underlay* network. This is shown in Figure 1.1.

It is often convenient to view a peer-to-peer system as a graph. In the graph, the nodes in the overlay are represented as vertices. Similarly, the neighbourhood relation of any two nodes in the overlay is represented as an edge in the graph. The shape of the graph depends on how the neighbours of a node are selected in the overlay. Based on the shape of the graph, peer-to-peer systems are classified into two broad categories: *unstructured* overlay networks and *structured* overlay networks. Figure 1.2 depicts this classification, which is explained in the following sections.



(a) An Unstructured Overlay Network (b) A Structured Overlay Network

Figure 1.2: An unstructured overlay network, and a structured overlay network with a ring geometry.

1.1.1 Unstructured Overlay Networks

As the name suggests, in unstructured overlay networks, there is no particular structure of the overlay, i.e. the graph induced by the nodes is unstructured. Gnutella and Kazaa are two popular examples of unstructured overlay networks currently being used on the Internet. In Gnutella, a node has random neighbours in the network that are changing all the time. To search for a data item, a node floods the network with the query by sending it to all its neighbours. Each node receiving the query forwards it to all its neighbours. Once the query reaches a node that has the requested target data item, the data item is transfered to the querying node. Normally, a query contains a time-to-live entry so that the flooding process terminates after a number of steps/hops.

The main disadvantages of this approach are hampered scalability and guarantees on finding the data item. The scalability is hampered because flooding the network with messages is costly, especially when there are millions of nodes. Similarly, when using a time-to-live, it might happen that the query terminates before reaching the node that had the sought data item.

1.1.2 Structured Overlay Networks

In structured overlay networks, a structure is induced by the edges of the graph representing the overlay, i.e. neighbour links of nodes. The structure is called the *geometry* of the structured overlay network. A structured overlay network utilizes an identifier space. Nodes are assigned identifiers from this space, and each node is responsible for cer-

tain identifiers. The basic operation that structured overlay networks offer is a *lookup* for an identifier. The result of a lookup for an identifier is the node responsible for the identifier. Structured overlay networks are the focus of this thesis; hence, we present more details of a structured overlay network with ring geometry in Chapter 2.

Structured overlay networks have the attractive property that starting from any node in the network, any other node is reachable in few steps (usually $O(\log(N))$, where N is the number of nodes in the system). Structured overlay networks have the additional desirable features of scalability and better guarantees of finding a published data item compared to unstructured overlay networks, while requiring a few number of neighbours per node.

Distributed Hash Tables (DHTs) are a popular data-structure built on top of structured overlay networks. As the name suggests, DHTs provide an abstraction to store data items under a key in the network. The data item can later be retrieved through the key that was used to store it. To achieve this, DHTs provide two operations; *put(k, v)*, to store a data item with value v under key k, and *get(k)*, to retrieve a data item stored with key k. Both put and get operations use the overlays lookup operation to reach the node responsible for serving the key k. The data item is then stored on/retrieved from the responsible node.

Routing Table Size	**Lookup Steps**	**Examples**
$O(\log N)$	$O(\log N)$	Chord [153] (ring), Pastry [127] (hybrid of ring & tree)
$O(d)$	$O(\log N)$	Koorde [73] (de Bruijn), Viceroy [103] (butterfly)
$2d$	$O(N^{\frac{1}{d}})$	CAN [124] (d-torus)
$O(\sqrt{N})$	$O(1)$	Kelips [61]

Table 1.1: Comparison of properties of selected structured overlays. Here, N is the size of the network.

Since structured overlay networks can have various geometries, different approaches emerged to build a structured overlay. These approaches differed in their geometry, sizes of routing tables, number of routing steps needed for a lookup, and number of steps required to incorporate changed in the network. Some of the popular structured overlay networks, with their properties, are listed in Table 1.1.

1.1.3 Gossip/Epidemic Algorithms

Gossiping is an important technique used in large-scale distributed systems to solve many problems. It has gained tremendous popularity in P2P systems because it is scalable, yet simple to use and robust to failures. In gossiping, information is spread in the network similar to the way a rumor is spread, where continuous exchange of a rumor between pairs of people results in its global spread. Gossip algorithms are also referred to as *epidemic* algorithms since in gossiping, information is spread in the system in a manner similar to the spread of a viral infection in a community.

Gossip algorithms are periodic, where in a period, each node chooses a random node in the system to gossip with. This gossip can be sending information only (push), receiving information only (pull) or an exchange of information (push-pull). It has been shown that given each node has access to random nodes in the system, gossiping can be used to spread an information to all nodes in $O(\log N)$ steps, where N is the size of the network [120].

Gossip algorithms were first used by Demers *et. al.* [39] in 1987. They employed gossiping with a technique called *anti-entropy* to maintain a replicated database. In their solution, whenever a replica receives any changes, it starts to gossip the changes with other replicas. This gossip spreads like an epidemic in the network. On receiving such a gossip, a replica can use anti-entropy to update its local state based on the changes mentioned in the gossip, and resolve any inconsistencies. Thus, the replicated database remains updated and consistent.

Gossip algorithms have since been used for solving various problems. We employ gossip techniques in this thesis for spreading information. Some of its usage in other P2P systems are:

1. Disseminating information to all nodes in the system, such as broadcasting a message [18].

2. Managing membership in an overlay to provide a node with continuous access to random nodes in the system [160, 69].

3. Computing aggregates of values locally stored at all nodes, such as average, summation, maximum, and minimum [70].

4. Fast bootstrapping [109] and maintaining routing tables in structured overlay networks [89, 61].

5. Clustering/ranking nodes with similar properties or preferences [129, 161]

1.1.4 Modern uses of Peer-to-peer Systems

Peer-to-peer systems are decentralized, which makes them scale better. Similarly, such systems are designed to self-manage under failures and joins of new nodes. Due to their scalability, self-management, and fault-tolerance, applications built using the peer-to-peer paradigm are extremely popular on the Internet. This is evident from a recent study which showed that peer-to-peer applications dominate the Internet usage [154], and are likely to continue to do so [44]. Content distribution, including file sharing and media streaming, are the main contributors to the peer-to-peer bandwidth usage.

File sharing systems, such as BitTorrent [28], eMule [41], and Gnutella, are widely used on the Internet. The Kad network, an implementation of the Kademlia structured overlay [106], provides the base of most of these file sharing systems and is estimated to have at least 2–4 million active users [150, 151]. Similarly, media streaming through peer-to-peer mechanisms is also very common since it does not require expensive servers. PPLive [122] and Sopcast [148] are such popular live video streaming peer-to-peer systems, with a reported 3.5 million daily active users of PPLive [65].

Modern web applications generate and access prodigious amounts of data, which requires the data storage to be scalable. This has led peer-to-peer mechanisms to be used inside data centers. Data center environments are more stable than the open Internet, as the machines and networking equipment are managed by the data center owners. Nevertheless, since data centers can contain hundreds of thousands of machines, features such as fault-tolerance and self-management are highly desirable which are the basis of peer-to-peer systems. Data stores, such as Dynamo [38], Cassandra [81], Voldemort [43], and Riak [13], employ peer-to-peer techniques and are widely used in the industry today, e.g. Amazon.com Inc. uses Dynamo, and NetFlix. Inc. uses Cassandra.

One of the most widely used Internet phone application, Skype [147], uses peer-to-peer principles as well. Skype has over 650 million registered users [149], with a record of 36 million concurrent active users [4]. Since peer-to-peer systems are decentralized, supporting such scales becomes easier. Another recent peer-to-peer system gaining attraction is Bitcoin [112]. Bitcoin is a peer-to-peer digital currency, with no central currency issuer, and nodes in the network regulate balances and transactions.

1.2 Research Objectives and Contributions

While structured overlay networks were designed for dynamic environments, and to be fault-tolerant, some issues pertaining to fault tolerance remained unsolved. This thesis focuses on fault tolerance on the *routing level* and the *data level* in structured overlay networks. The routing level is concerned with the routing tables of the nodes in the peer-to-peer system, which need to be updated due to any network or node failures. The data level is related to any data stored within the peer-to-peer system. Next, we list the research issues and contributions that are the focus of this thesis.

Routing level: On the routing level, we address the following problems:

- An underlying network partition can partition an overlay into separate independent overlays. Once the partition ceases, the overlays should be merged as well. We address the problem of detecting such underlying network partitions and mergers, and merging multiple overlays into one.

- To be able to bootstrap, maintain the overlay, and merge multiple overlays, separate algorithms are needed to handle each scenario. While using multiple algorithms to achieve a single goal can be complicated, it is also error prone as the effects of one algorithm on the others have to be properly understood. At this end, we attack the challenge of having a single algorithm for bootstrapping, overlay maintenance, and handling network partitions and mergers.

- Overlays operate over dynamic environments, where nodes join and leave the system all the time. An estimate of the network size is useful in many scenarios, such as load-balancing, adjusting routing table sizes, and tuning the rate of routing table maintenance. We solve the problem of estimating the current network size in ring-based structured overlay networks.

- Due to the dynamic set of participating nodes, and asynchronous networks, multiple requests (lookups) to find an item in an overlay can end up with different results. We explore the frequency of such lookup inconsistencies, and propose mechanisms to reduce them in structured overlay networks.

Data level: On the data level, we address the following research challenges:

- For fault tolerance amid network and node failures, data storage systems built on top of overlays replicate each data item on a set of nodes (replicas). Such storage systems do not guarantee strong data consistency across the replicas. In this thesis, we address the problem of achieving data consistency and partition tolerance in a scalable and completely decentralized setting using overlays.

- Existing replication schemes for structured overlays are sensitive to node joins, leaves, and failures, resulting in a high number of reconfigurations of replication groups. We discuss the shortcomings of existing replication schemes for overlays, and propose a solution.

Next, we provide a summary of our contributions for each problem area addressed in this thesis work. These contributions have been published [134, 136, 137, 135, 140, 141, 139, 22, 133, 138], and are the focus of the next chapters.

1.2.1 Handling Network Partitions and Mergers

In our work, we motivate that handling underlying network partitions and mergers is a core requirement for structured overlays. We argue that since fault-tolerance, scalability, and self-management are the basic properties of overlays, they should tolerate network partitions and mergers.

Our contribution is two-fold [134, 136]. First, we propose a mechanism for detecting a scenario where a partition occurred and later, the underlying network merged. Second, we propose two algorithms for merging overlays, *simple ring unification* and *gossip-based ring unification*. Simple ring unification is a low-cost solution with respect to the number of messages sent (message complexity), yet it suffers from two problems: (1) slow convergence time ($O(N)$ time for a network size of N), and (2) less robustness to churn.

Gossip-based ring unification addresses both short-comings of simple ring unification, i.e. it has a high convergence rate ($O(\log N)$ time for a network size of N), and is robust to churn, yet it is a high-cost solution in terms of message complexity. In our solution, we provide a *fanout* parameter that can be used to control the trade-off between message and time complexity in gossip-based ring unification.

1.2.2 Bootstrapping, Maintenance, and Mergers

Our ring unification algorithms act as add-ons to an overlay maintenance algorithm. They are started when a network merger is detected, and terminate once the overlays are merged into a single overlay. We argue that apart from dealing with normal churn rates, handling extreme scenarios – such as bootstrapping, network partitions and mergers, and flash crowds – is fundamental to providing a fault-tolerant and self-managing system, and thus, structured overlay networks should intrinsically be able to handle them.

In this thesis, we present *ReCircle* [138], an overlay algorithm that other than being able to perform periodic maintenance to handle churn like any other overlay, can merge multiple structured overlay networks. We show how such an algorithm can be used for decentralized bootstrapping, which is an important self-organization requirement that has been ignored by structured overlay networks. ReCircle does not have any extra cost during normal maintenance compared to an isolated overlay maintenance algorithm. Furthermore, the algorithm is tunable to trade bandwidth consumption for lower convergence time during extreme events like bootstrapping and handling network partitions and mergers. We designed ReCircle to be reactive to extreme events so that it can converge faster when such events occur.

1.2.3 Network Size Estimation

Gossip-based aggregation [71] is known to be a highly accurate method of estimating the current network size of an overlay [107]. In our work, we discuss the shortcomings of gossip-based aggregation for network size estimation. We argue that the main disadvantage of gossip-based aggregation is that a failure of a single node early on, can severely affect the final estimate. Furthermore, gossip-based aggregation requires predefining the convergence time t for the estimation. This may lead to inaccurate estimation of the network size if t is shorter than necessary, or delay the estimate from being used if t is longer than necessary.

Our contribution is an aggregation-based solution [135] that provides an estimate of the current network size for ring-based overlays, and does not suffer the shortcomings of aggregation by Jelasity *et. al.* [71]. We evaluate our solution extensively and show its effectiveness under churn and for various network sizes.

1.2.4 Lookup Inconsistencies

In our work, we argue that it is nontrivial to provide consistent data services on top of structured overlays since key/identifier lookups can return inconsistent results. We study the frequency of occurrence of such lookup inconsistencies. We propose a solution to reduce lookup inconsistencies by assigning responsibilities of key intervals to nodes. As a side effect, our solution may lead to unavailability of keys. Thus, we present our results as a trade-off between consistency and availability [140, 141, 139].

Since many distributed algorithms employ quorum techniques, we extend our work by analyzing the probability that majority-based quorum techniques will function correctly in overlays in spite of lookup inconsistencies. We present a theoretical model of measuring the number of lookup inconsistencies while using replication. We show that apart from inconsistencies arising from churn, a major contributor to lookup inconsistencies is the inaccuracy of failure detectors. Hence, special attention should be paid while designing and implementing a failure detector.

1.2.5 Data Consistency

Due to the scalability and self-management features of structured overlay networks, large-scale data stores, e.g. Cassandra [81], and Dynamo [38], are built on top of overlays. These storage systems target applications that do not require strong data consistency, but instead focus on availability. While such data stores are scalable and easy to manage, there are numerous applications that require strong data consistency guarantees.

Our contribution is *consistent quorums* [22]; an approach to guarantee linearizability [63] – the strongest form of data consistency – in a decentralized, self-organizing, and dynamic asynchronous environment. As a showcase, we use consistent quorums to build *CATS*, a partition-tolerant, scalable, elastic, and self-organizing key-value store that provides linearizability. We evaluate CATS under various workloads, and show that it is scalable and elastic. Furthermore, we evaluate the cost of achieving linearizability in CATS, which shows that the overhead is modest (5%) for read-intensive workloads.

1.2.6 Replication

We discuss popular replication techniques in structured overlay networks, including *successor-list replication* [153] and *symmetric replication*

[48], and their drawbacks. We show that successor-list replication is highly sensitive to churn; a single node join or failure event results in updating multiple replication groups. Furthermore, successor-list replication is inherently difficult to load-balance. Finally, successor-list replication is less secure and presents a bottleneck since there is a master replica of each replication group and all requests for that group have to go through the master replica. Similarly, symmetric replication requires a complicated bulk operation [47] for retrieving all keys in a given range when a node joins or fails.

Our contribution is *ID-Replication* [133], a replication scheme for structured overlays that does not suffer from the aforementioned drawbacks. It does not require requests to go through a particular replica. ID-Replication gives more control to an administrator and allows easier implementation of policies, without hampering self-management. Furthermore, ID-Replication allows different replication degrees for different key ranges. This allows for using higher number of replicas for hot spots and critical data. Our evaluation shows that ID-Replication is less sensitive to churn, thus better suited to be used for asynchronous networks where false failure detections are the norm. Since we use a generic design, ID-Replication can be used in any structured overlay network.

1.3 Organization

This thesis is organized as follows. Chapter 2 provides a background to the thesis. Chapters 3 and 4 present solutions on the routing level. Chapter 3 presents the motivation for handling network partitions and mergers in overlays, and discusses a mechanism of detecting when a network partition heals. It then provides various algorithms for merging multiple ring-based overlay networks into one. In Chapter 4, we present an algorithm for estimating the number of nodes in a peer-to-peer network, amid continuous churn.

Chapter 5 can be viewed as a bridge between the routing level and the data level. It discusses anomalies in routing pointers that can result in inconsistencies on the data level. It then presents techniques on the routing level to reduce data inconsistencies.

Chapters 6 and 7 deal with the data level. Chapter 6 presents a key-value store that is both, strongly consistent and partition tolerant. Next, we present a replication scheme for structured overlays in Chapter 7.

We conclude and present potential future directions in Chapter 8.

CHAPTER 2

Preliminaries

This thesis focuses on ring-based structured overlay networks. Next, we motivate this choice. Thereafter, we briefly discuss a model for a ring-based overlay used in this thesis. As an example, we discuss the Chord [153] overlay, which has a ring geometry. We describe how Chord maintains a ring topology amid node joins and failures. We then discuss techniques used in overlays on the data level, including replication schemes and maintaining consistency amongst the replicas.

Motivation for the Unidirectional Ring Geometry In this thesis work, we confine ourselves to unidirectional ring-based overlays, such as Chord [153], SkipNet [62], DKS [47], Koorde [73], Mercury [15], Symphony [104], EpiChord [90], and Accordion [92]. We believe that our algorithms can easily be adapted to other ring-based overlays, such as Pastry [127]. For a more detailed account on directionality and structure in overlays, we refer the reader to Onana *et al.* [8] and Aberer *et al.* [2].

The reason for confining ourselves to ring-based overlays is twofold. First, ring-based overlays constitute a majority of the existing overlays. Second, Gummadi *et al.* [58] diligently compared the geometries of different overlays, and showed that the ring geometry is most resilient to failures, while it is just as good as the other geometries when it comes to proximity. To simplify the discussion and presentation of our algorithms, we use notation that indicates the use of the Chord [153] overlay. But the ideas are directly applicable to all unidirectional ring-based overlays.

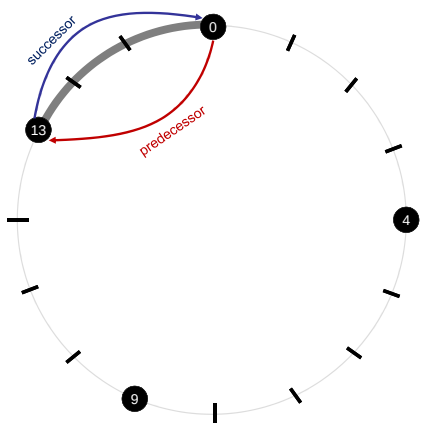


Figure 2.1: A ring-based overlay, with an identifier size of 16. Node 13 is the predecessor of 0, and it has 0 as its successor. Node 0 is responsible for the identifiers between 13 (exclusive) and 0 (inclusive), i.e. 14, 15, and 0.

2.1 The Routing Level

This section gives a model of a structured overlay used in this thesis, which is based on the principles of *consistent hashing* [76], and discusses routing level techniques.

2.1.1 A Model of a Ring-based Overlay

A ring-based overlay makes use of an *identifier space*, which for our purposes is defined as a set of integers $\{0, 1, \cdots, \mathcal{N}-1\}$, where $\mathcal{N}$ is some apriori fixed, large, and globally known integer. This identifier space is perceived as a ring that wraps around at $\mathcal{N}-1$. This is shown in Figure 2.1, where $N = 16$.

Every node in the system has a unique identifier from the identifier space. Node identifiers are typically assumed to be uniformly distributed on the identifier space. Each node keeps a pointer, *succ*, to its *successor* on the ring. The successor of a node with identifier p is the first node found going in clockwise direction on the ring starting at p. Similarly, every node also has a pointer, *pred*, to its *predecessor* on the ring. The predecessor of a node with identifier q is the first node met going in anti-clockwise direction on the ring starting at q. A *successor-*

list is also maintained at every node r, which consists of r's c immediate successors, where c is typically set to $\log_2(N)$, where N is the network size.

The identifier space is also used for partitioning tasks among nodes in the overlay. For instance, in *key-value stores* and Distributed Hashtables (DHTs) that use an overlay to store data items, the identifier space is used to partition the data items amongst nodes. Each data item is assigned an integer/identifier from the identifier space, called the *key* of the data item. Nodes in the overlay are responsible for storing data items that have keys in the vicinity of the node's identifier. For instance, in Chord, a node with identifier p is *responsible* for storing data items with keys $k \in (p.pred, p]$, i.e. all keys between p's predecessor (exclusive) and p (inclusive) going clockwise. We use the notation $k_{(a,b]}$ to denote the key range $(a, b]$, i.e., all $keys \in (a, b]$.

2.1.2 Maintaining Routing Pointers

In this thesis, we use *event-based notation* for presenting our algorithms since it models an asynchronous distributed system closely. In event-based notation, an algorithm is specified as a collection of *event handlers*. An event handler is defined by: an *event type*, parameters that define the contents of the event, the sender, and recipient of the event. Upon receiving an event of a certain type, its event handler is executed. While processing an event in the event handler, a node can communicate with other nodes by sending events.

As discussed in Section 1.1, each node in an overlay has a set of routing pointers, called *routing table*. Since overlays operate over dynamic environments, routing pointers get outdated upon node joins and failures. The goal of an overlay maintenance algorithm is to handle and incorporate any dynamism in the system. This goal is achieved by updating routing pointers to reflect the changes in the system.

Chord [153] handles joins and leaves/failures using an overlay maintenance protocol called *periodic stabilization*, shown as Algorithm 1 in event-based notation. The essence of the protocol is that each node p periodically attempts to find and update its successor to a node which is closer (clock-wise) to p than p's current successor. Similarly, each node sets its predecessor to a node closer (anti-clockwise) than its current predecessor. In Algorithm 1, this is done as follows.

Each node periodically (every γ time units) sends a `WhoIsPred` event to its successor (line 2). Upon receiving such an event (line 4), a node replies by sending an event of type `WhoIsPredReply`, with its predecessor as a parameter of the event. When a node p receives an event of

Algorithm 1 Chord's Periodic Stabilization [153]

```
 1: every δ time units at n
 2:     sendto succ : WhoIsPred⟨⟩
 3: end event

 4: receipt of WhoIsPred⟨⟩ from m at n
 5:     sendto m : WhoIsPredReply⟨pred⟩
 6: end event

 7: receipt of WhoIsPredReply⟨succPred⟩ from m at n
 8:     if succPred ∈ (n, succ) then
 9:         succ := succPred
10:     end if
11:     sendto succ : Notify⟨⟩
12: end event

13: receipt of Notify⟨⟩ from m at n
14:     if pred = nil or m ∈ (pred, n) then
15:         pred := m
16:     end if
17: end event
```

type `WhoIsPredReply` with parameter *succPred* (line 7), p sets *succPred* as its successor if *succPred* is closer to p than $p.succ$ when going clockwise starting at p. Thereafter, p notifies its successor about its presence by sending a `Notify` event (line 11). Upon receiving such a notification (line 13) from a sender s, a node q sets s as its predecessor if either s is closer to q than $q.pred$, q's current predecessor, going anti-clockwise from q, or if q does not have a valid predecessor.

Leaves and failures are handled by having each node periodically check whether its predecessor *pred* is alive, and setting $pred := nil$ (invalid predecessor) if it is found dead. Moreover, each node periodically checks to see if its successor *succ* is alive. If it is found to be dead, it is replaced by the closest alive successor in the successor-list.

Joins are also handled periodically. A joining node makes a lookup to find its successor s on the ring, and sets $succ := s$. The rest is taken care of by periodic stabilization as follows. Each node periodically asks for its successor's *pred* pointer, and updates *succ* if it finds a closer successor. Thereafter, the node notifies its current *succ* about its own existence, such that the successor can update its *pred* pointer if it finds that the notifying node is a closer predecessor than *pred*. Hence, any joining

node is eventually properly incorporated into the ring.

Lookup

A *lookup* for an identifier *id* initiated at any node in the system is a request to find the node responsible for *id*, i.e. node p such that $id \in (p.pred, p]$. Here, we say that the lookup(*id*) resolves to p. Applications built on top of overlays can use the lookup service provided by the overlay. For instance, DHTs use the lookup service to store data and provide a *put*/*get* interface for scalable distributed storage. A `put(key, value)` operation initiates lookup(*key*), and stores *value* on the node that the lookup resolves to. Similarly, a `get(key)` operation initiates lookup(*key*), and returns the data stored against the key at the node that the lookup resolves to.

Using successor pointers, a lookup request can be resolved in $O(N)$ hops, where N is the number of nodes in the system, by forwarding the lookup clockwise. This is shown in Algorithm 2, where ClosestPrecedingNeighbour(*id*) returns the successor. Ring-based overlays maintain additional routing pointers on top of the ring to enhance such routing requests. These additional routing pointers constitute the *routing table* of nodes and are used to perform greedy routing to reduce the number of hops when resolving lookups. For instance, in Chord, nodes maintain additional routing pointers, called *fingers*, that are exponentially spread on the identifier space. Concretely, each node p keeps a pointer to the successor of the identifier $p + 2^i \pmod{\mathcal{N}}$ for $0 \leq i < \log_2(N)$. Nodes use these fingers to resolve lookups for identifiers by performing greedy routing. In Algorithm 2, greedy routing is achieved by ClosestPrecedingNeighbour(*id*) returning the closest finger that precedes identifier *id*. Our algorithms in this thesis are independent of the scheme for placing these additional routing pointers.

Algorithm 2 A Lookup Operation

```
1: receipt of Lookup⟨id⟩ from m at n
2:     if id ∈ (n, succ) then
3:         sendto m : LookupResult⟨succ⟩
4:     else
5:         forwardTo := ClosestPrecedingNeighbour(id)
6:         sendto forwardTo : Lookup⟨id⟩
7:     end if
8: end event
```

2.2 The Data Level

In this section, we provide an overview of techniques used on the data level. First, we introduce various replication schemes proposed for overlays. Next, we introduce a class of algorithms used for maintaining consistency amongst the replicas.

2.2.1 Replication

Similar to other distributed systems, data fault-tolerance and reliability in structured overlays is achieved via replication. Various strategies for replication in overlays have been proposed, such as successor-list replication [153], symmetric replication [48], and using multiple hash functions [124, 164]. In the multiple hash functions scheme, a data item with key k is stored under multiple keys, which are calculated by hashing k using different hash functions. Such a scheme is known as *key-based* replication. Next, we discuss successor-list replication and symmetric replication.

Successor-list replication

As discussed in Section 2.1.1, each node q is responsible for storing keys between q's predecessor and q. For a replication degree of r in successor-list replication, a key k is stored on the node q that is responsible for storing k, and $r-1$ immediate successors of q. In Figure 2.2, node 30 is responsible for storing keys $k \in (20, 30]$, and k are replicated on $\{30, 35, 40\}$, which is called the *replica group* for k. As nodes join and leave the system, the successor, predecessor and successor-lists are updated, leading to changes in the replica groups and respective transfer of data between nodes. Since the responsibility range of a node is the unit of replication, successor-list replication is an instance of a *node-based* replication.

Symmetric replication

Symmetric replication is an instance of key-based replication, where each identifier is related to r other identifiers based on a certain relation. Such a relation is symmetric, i.e., if an identifier i is related to j, then j is related to i as well. Since each identifier is symmetrically related to r identifiers, it creates identifier groups of size r that can be used for replication. In essence, the identifier space is partitioned into $\frac{N}{r}$ equivalence classes, and identifiers/keys in an equivalence class are related to each other. Each equivalence class is a replication group, and

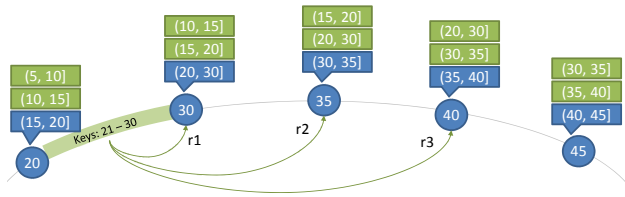


Figure 2.2: Successor-list replication with replication degree 3. The replication group for *keys* $\in [21, 30]$ is $\{30, 35, 40\}$. Similarly, responsibility of node 35, i.e. $(30, 35]$, is replicated on 3 nodes encountered clockwise from 35, i.e. 35, 40 and 45.

each replica can be located by making a lookup for the identifier of the replica.

2.2.2 Consistency and Quorum-based Algorithms

While replication provides fault-tolerance, and possibly improved performance, it introduces the problem of achieving consistency of the data stored on the replicas. Many consistency models have been proposed over the years, ranging from weaker consistency models, such as, eventual consistency [157], read-your-own writes, causal consistency [85], sequential consistency [83], to stronger consistency models, such as linearizability [84, 63], one-copy serializability, and strict-serializability [117] transactions.

To achieve fault-tolerance and consistency, *quorum-based* algorithms are most widely used, e.g. for replicated data [50], consensus [86], replicated state machines [132], concurrency control [158], atomic shared-memory registers [11], and non-blocking atomic commit [56]. The basic idea of quorum-based algorithms is that conflicting operations acquire a sufficient number of votes from different replicas such that they have at least one intersection at one replica. Gifford introduced an algorithm for the maintenance of replicated data that uses weighted votes [50]. Every replica has a certain number of votes. Each read operation has to collect r votes and each write operation has to collect w votes, where $r + w$ exceeds the number of votes assigned to a data item. Thus, every read quorum and every write quorum overlap in at least one replica.

Without loss of generality, we focus on quorum-based algorithms where a quorum constitutes of a majority of replicas, and each replica is assigned exactly one vote. Here, each operation has to be performed on a majority of replicas. For instance, a `put(key, value)` is considered

incomplete until the value has been stored on a majority of the replicas. If the set of replicas remains static, all operations overlap on at least one node, thus guaranteeing that all operations will use/see the latest data.

Eventual consistency: The most popular consistency model provided by key-value stores built on top of overlays is eventual consistency [157, 39, 159]. For instance, Dynamo [38], Cassandra [81], and Riak [13], all provide eventual consistency. Eventual consistency means that for a given key, data values may diverge at different replicas, e.g., as a result of operations accessing less than a quorum of replicas or due to network partitions [19, 51]. Eventually, when the application detects conflicting replicas, it needs to reconcile the conflict. The merge mechanisms for resolving conflicts is dependent on the application semantics.

Linearizability: The strongest form of consistency for a single data item read/write operations is linearizability [63], also known as *atomic consistency*. For a replicated storage service, linearizability provides the illusion of a single storage server: each operation applied at concurrent replicas appears to take effect instantaneously at some point between its invocation and its response. Linearizability guarantees that a read operation ($get(k)$ for a key in overlays) always returns the value updated by the most recent write/update operation ($put(k, v)$ in overlays), or the value of a concurrent write, and once a read returns a value, no subsequent read can return an older/stale value. Such semantics give the appearance of a single global consistent memory. Attiya et al. [11] present a quorum-based algorithm that satisfies linearizability in a fully asynchronous message-passing system. Linearizability is composable. In a DHT, this means that if operations on each individual key-value pair are linearizable, then all operations on the whole key-value store are linearizable, and the DHT as a whole can be termed as linearizable.

Consistency, Availability, and Partition-tolerance: The CAP theorem [19, 51], also known as Brewer's conjecture, states that in a distributed asynchronous network, it is impossible to achieve consistency, availability and partition tolerance at the same time. Hence, system designers have to chose two out of the three properties. Such a choice depends on the target application. Network partitions are a fact of life, hence applications in general chose between consistency and availability. For some applications, availability is of utmost importance, while weaker consistency guarantees suffice. Yet other applications sacrifice availability for strong consistency.

CHAPTER 3

Network Partitions, Mergers, and Bootstrapping

An underlying network partition can partition an overlay into separate independent overlays. Once the partition ceases, the overlays should be merged as well. In this chapter, we discuss the challenges of handling network partitions and mergers in overlays on the routing level. We provide a solution that merges multiple overlays into one, and terminates after wards. We extend our solution to provide a non-terminating algorithm that can bootstrap an overlay efficiently, maintain the overlay, and handle extreme scenarios such as network partitions and mergers.

3.1 Handling Network Partitions and Mergers

Structured overlay networks and DHTs are touted for their ability to provide scalability, fault-tolerance, and self-management, making them well-suited for Internet-scale distributed applications. As these applications are long lived, they will always come across network partitions. Since overlays are known to be fault-tolerant and self-manage, they have to be resilient to network partitions.

Network partitions are a fact of life. Although network partitions are not very common, they do occur. Hence, any long-lived Internet-scale system is bound to come across network partitions. A variety of

reasons can lead to such partitions. A WAN link failure, router failure, router misconfiguration, overloaded routers, congestion due to denial of service attacks, buggy software updates, and physical damage to network equipment can all result in network partitions [116, 21, 118, 156]. Similarly, natural disasters can result in Internet failures. This was observed when an earthquake in Taiwan in December 2006 exposed the issue that global traffic passes through a small number of seismically active "choke points" [115]. Several countries in the region connect to the outside world through these choke points. A number of similar events leading to Internet failures have occurred [21]. On a smaller scale, the aforementioned causes can disconnect an entire organization from the Internet [116], thus partitioning the organization.

Apart from software and hardware failures, political and service provider policies can also result in network partitions. For instance, the disputation between two Tier-1 ISPs, *Level 3* and *Cogent*, lead to inaccessibility across the two networks for three weeks [27]. A similar instance lead to network breakage for a large number of customers across the Atlantic when *Cogent* and *Telia* had a two week disagreement [67]. Similarly, due to government policies, the Internet was cut-off in Egypt for more than 24 hours resulting in the network of the whole country being partitioned away [16].

The importance of the problem of handling network partitions has long been known in other domains, such as those of distributed databases [37] and distributed file systems [157]. Since the vision of structured overlay networks is to provide fault-tolerance and self-management at large-scale, we believe that structured overlay networks should be able to deal with network partitions and mergers. On the same lines, while deploying an application built on top of a structured overlay, the first major problem reported and strongly suggested to be solved by Mislov *et al.* [108] was:

> "A reliable decentralized system must tolerate network partitions."

While deploying ePOST [108], a decentralized email service build on top on the Pastry [128] structured overlay, on PlanetLab, Mislove *et. al.* recorded the number of partitions experienced over a period of 85 days. Figure 3.1 shows their results, which clearly suggests that partitions occur all the time over the Internet. Thus, as any other Internet-scale system, structured overlays should be able to handle underlying network partitions and mergers.

Apart from network partitions, the problem of merging multiple overlays is interesting and useful in itself. Decentralized bootstrapping

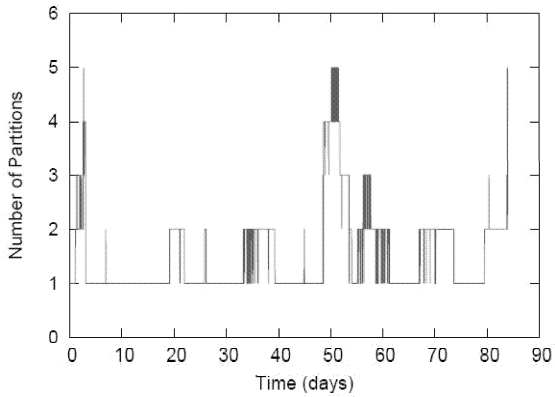


Figure 3.1: Number of connected components observed over an 85 day period on PlanetLab. Figure taken from ePOST [108].

of overlays [34] can be done by building overlays separately and independently, regardless of existing overlays. Later, these overlays can be merged into one. Also, it might be that overlays build independently later have to be merged due to overlapping interests.

Consequently, a crucial requirement for practical overlays is that they should be able to deal with network partitions and mergers.

Some overlays cope with network partitions, but none can deal with network mergers. This is because a network partition, as seen from the perspective of a single node, is identical to massive node failures. Since overlays have been designed to cope with churn (node joins and failures), they can self-manage in the presence of such partitions. For instance, as periodic stabilization in Chord can handle massive failures [96], it also recovers from network partitions, making each component of the partition eventually form its own ring. We have confirmed such scenarios through simulation. However, most overlays cannot cope with network mergers.

The merging of overlays gives rise to problems on two different levels: *routing level* and *data level*. The routing level is concerned with healing the routing information after a partition ceases. The data level is concerned with the consistency of the data items stored in the overlay. In this thesis, we address issues at both, routing and data, levels as follows:

- In Section 3.1.1, we discuss how nodes in an overlay can detect a

network merger after the network partition has ceased. Once a node detects a network merger, it can trigger an overlay merger algorithm.

- On the routing level, we show how to merge multiple ring-based overlays into one overlay, effectively fixing the successor and predecessor pointers. Our solution, known as *Ring-Unification* [134, 136] and presented in Section 3.2, can be triggered once a node detects a network merger or an administrator initiates a merger of independent overlays, and terminates once the overlays have been merged.

- In Section 3.3, we extend ideas from Ring-Unification and Periodic Stabilization, and present a non-terminating overlay maintenance algorithm, called *ReCircle* [138]. ReCircle can be used as the sole overlay algorithm for maintaining the geometry of the overlay under normal levels of churn as well as extreme levels, e.g. that arise due to network partitions and mergers. Furthermore, ReCircle can be used for efficiently bootstrapping an overlay.

- On the data level, we show how to achieve consistency amid network partitions and mergers. We present a key-value store built on top of ReCircle, called *CATS*, in Chapter 6. CATS provides linearizability [63], the strongest form of consistency, and is partition tolerant. Since CATS is built on top of an overlay, it is scalable, self-managing, completely decentralized, and works in dynamic asynchronous environments.

3.1.1 Detecting Network Partitions and Mergers

A network partition results in the overlay to be divided into multiple independent overlays. We confirmed this behaviour via simulations for the Chord overlay. After a network partition, each partitions forms its own ring. The problem unsolved is that once the partition ceases, the overlay remains divided and each overlay ring continues to operate independently. In this section, we discuss how to detect that a prior network partition has ceased. Once the system detects that the partition has healed, it can trigger an overlay merger algorithm, such as Ring-Unification (Chapter 3.2) and Recircle (Chapter 3.3), to merge the overlays into a single overlay. Without loss of generality, we consider the case where an overlay is partitioned into two overlays for simplicity.

For two rings to be merged, at least one node needs to have knowledge about at least one node in the other ring. This is facilitated by

the use of *passive lists*. Whenever a node detects that another node has failed, it puts the failed node, with its routing information[1] in its passive list. Every node periodically pings nodes in its passive list to detect if a failed node is alive again. When this occurs, it can trigger a ring merging algorithm. A network partition will result in many nodes being placed into passive lists. When the underlying network merges, nodes will be able to ping nodes in their passive list. Thus, the merger will be detected and rectified through the execution of a ring merging algorithm.

The detection of an alive node in a passive list does not necessarily indicate the merger of a partition. It might be the case that a single node is incorrectly detected as failed due to a premature timeout of a failure detector. Thus, the ring merging algorithm should be able to cope with this by trying to ensure that such false-positives will terminate the algorithm quickly. It might also be the case that a previously failed node rejoins the network, or that a node with the same overlay and network address as a previously failed node joins the ring. Such cases are dealt with by associating with every node a globally unique random *nonce*, which is generated each time a node joins the network. Hence, if the algorithm detects that a node in its passive list is again alive, it can compare the node's current nonce value with that in the passive list to avoid a false-positive, as that node is likely a different node that coincidentally has the same overlay and network address.

An alternative mechanism to avoid adding nodes to the passive list when the network has not partitioned is by using a network size estimation algorithm, such as the one given in Chapter 4. If the size of the network changes abruptly, it is an indication that a partition has occurred. In such cases, even if one partition is larger than the other, it is sufficient for the nodes in the smaller partition to record the partition by populating their passive lists with failed nodes. Using the sudden change in the estimated network size as an indication of a network partition can reduce false positives.

A ring merging algorithm can also be invoked in other ways than described above. For example, it could occur that two overlays are created independently of each other, but later their administrators decide to merge them due to overlapping interests. It can also be that a network partition lasts very long. Since the size of passive lists are bounded, if a partition lasts long enough, nodes in passive lists from other partitions might be evicted/replaced. If the partition lasts long enough, it

[1] By routing information we mean a node's overlay identifier, network address, and nonce value (explained shortly).

can also happen that all nodes in the rings have been replaced, making the contents of the passive lists useless. In cases such as these, a system administrator can manually insert an alive node from another ring into the passive list of any of the nodes. The ring merger algorithm will take care of the rest.

3.2 Ring-Unification: Merging Multiple Overlays

This section focuses on the routing level, and presents mechanisms for merging multiple ring-based structured overlays. The routing level is concerned with fixing the routing information after a merger. Overlay merger algorithms presented in this chapter can be triggered by mechanisms discussed in Section 3.1.1. Given a solution to the problem at the routing level, it is generally known how to achieve weaker types of data consistency, such as eventual consistency [157, 39]. We present a solution for achieving strong consistency amid network partitions and mergers, at the cost of availability, in Chapter 6.

3.2.1 Ring Merging

Due to the large number of nodes involved in a peer-to-peer system, an overlay merging algorithm should be scalable, and avoid overloading the nodes and congesting the network. It should also be able to handle churn (node joins, failures, and leaves) as dynamism is common in such systems. Furthermore, an *efficient* solution for merging multiple overlays should minimize two metrics: (1) the time taken to converge the overlays into one (time complexity), and (2) the bandwidth consumption (message and bit complexity).

In this section, we present two algorithms for merging ring-based overlays. The first algorithm is low-cost, in terms of bandwidth consumption, yet has slow convergence rate and less resilient to churn. Our second algorithm is more robust to churn and allows the system designer to adjust, through a *fanout* parameter, the tradeoff between bandwidth consumption and time it takes for the algorithm to complete. Through evaluation, we show typical fanout values for which our algorithm completes quickly, while keeping the bandwidth consumption at an acceptable level.

For two or more rings to be merged, at least one node needs to have knowledge about at least one node in another ring. This is facilitated by the use of passive lists (Section 3.1.1). The detection of an alive node in a passive list does not necessarily indicate the merger of an underlying network partition. Thus, a ring merging algorithm should be able

to cope with this by trying to ensure that such false-positives will terminate the algorithm quickly. Apart from using passive lists to detect partitions and mergers, a system administrator can manually initiate merger of multiple overlays by inserting an alive node from another ring into the passive list of any of the nodes. The ring merger algorithm takes care of the rest.

3.2.2 Simple Ring Unification

In this section, we present the simple ring unification algorithm (Algorithm 3). As we later show, the algorithm will merge the rings in $O(N)$ time for a network size of N. Later, we show how the algorithm can be improved to make it complete the merger in substantially less time.

Algorithm 3 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
```

Algorithm 3 makes use of a queue maintained at each node called

detqueue, which will contain any alive nodes found in the passive list. The queue is periodically checked by every node p, and if it is non-empty, the first node q in the list is picked to start a ring merger. Ideally, p and q will be on two different rings. But even so, the distance between p and q on the identifier space might be very large, as the passive list can contain any previously failed node. Hence, the event mlookup(id) is used to get closer to id through a lookup. Once mlookup(id) gets near its destination id, it triggers the event trymerge(a, b), which tries to do the actual merging by updating *pred* and *succ* pointers to a and b respectively.

The event mlookup(id) is similar to a Chord lookup, which tries to do a greedy search towards the destination id. One difference is that it terminates the lookup if it reaches the destination and locally finds that it cannot merge the rings. More precisely, this happens if mlookup(id) is executed at id itself, or at a node whose successor is id. If an mlookup(id) executed at n finds that id is between n and n's successor, it terminates the mlookup and starts merging the rings by calling trymerge. Another difference between mlookup and an ordinary Chord lookup is that an mlookup(id) executed at n also terminates and starts merging the rings if it finds that id is between n's predecessor and n. Thus, the merge will proceed in both clockwise and anti-clockwise direction.

The event trymerge takes as parameters a candidate predecessor, $cpred$, and a candidate successor $csucc$, and attempts to update the current node's *pred* and *succ* pointers. It also makes two recursive calls to mlookup, one towards $cpred$, and one towards $csucc$. This recursive call attempts to continue the merging in both directions. Figure 3.2 shows the working of the algorithm.

In summary, mlookup closes in on the target area where a potential merger can happen, and trymerge attempts to do local merging and advancing the merge process in both directions by triggering new mlookups.

3.2.3 Gossip-based Ring Unification

The simple ring unification presented in the previous section has three disadvantages. First, it is slow, as it takes $O(N)$ time to complete the ring unification. Second, it cannot recover from certain pathological scenarios. For example, assume two distinct rings in which every node points to its successor and predecessor in its own ring. Assume furthermore that the additional pointers of every node point to nodes in the other ring. In such a case, an *mlookup* will immediately leave the initiating node's ring, and hence may terminate. We do not see how such a pathological scenario could occur due to a partition, but the *gossip-based*

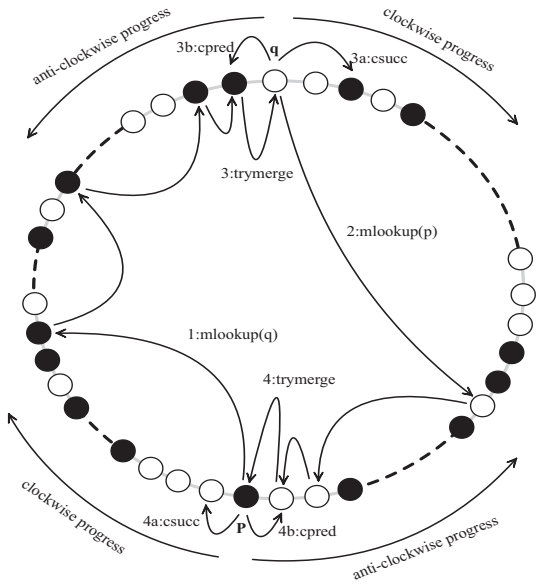


Figure 3.2: Filled circles belong to overlay 1 and empty circles belong to overlay 2. The algorithm starts when p detects q, p makes an mlookup to q and asks q to make an mlookup to p.

ring unification algorithm (Algorithm 4) rectifies both disadvantages of the simple ring unification algorithm. Third, the simple ring unification is less robust to churn, as we discuss in the evaluation section.

Algorithm 4 is, as its name suggests, gossip-based. The algorithm is essentially the same as the simple ring unification algorithm, with a few additions. The intuition is to have the initiator of the algorithm immediately start multiple instances of the simple algorithm at random nodes, with uniform distribution. But since the initiator's pointers are not uniformly distributed, the process of picking random nodes is incorporated into mlookup. Thus, mlookup(id) is augmented so that the current node randomly picks a node r in its current routing table and starts a ring merger between id and r. This change alone would, however, consume too much resources.

Two mechanisms are employed to prevent the algorithm from consuming too many messages, which could give rise to positive feedback cycles that congest the network. First, instead of immediately triggering an mlookup at a random node, the event is placed in the corresponding node's *detqueue*, which is only checked periodically. Second, a constant

Algorithm 4 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
```

number of random mlookups are created. This is regulated by a fanout parameter called F. Thus, the fanout is decreased each time a random node is picked, and the random process is only started if the fanout is larger than 1. The *detqueue*, therefore, holds tuples, which contain a node identifier and the current fanout parameter. Similarly, mlookup takes the current fanout as a parameter. The rate for periodically checking the *detqueue* can be adjusted to control the rate at which the algorithm generates messages.

3.2.4 Discussion

Ring-unification is a terminating algorithm, i.e. it terminates once the overlays have been merged into one. Thus, it has been designed such that it can be used as an add-on to an existing overlay maintenance algorithm, e.g. Chord. In Section 3.3, we present a non-terminating algorithm that is responsible for fast and efficient bootstrapping, overlay maintenance, and handling network partitions and mergers.

Lookups made after the merge is complete perform normally. An interesting issue is the behaviour of lookups made during the merger of the overlays. Such lookups may not always succeed in finding the related data item. For instance, say there are two overlays to be merged, O_1 and O_2, and a data item is stored with key k in O_1. Consider a node n initially part of O_1, and a node m initially part of O_2. During the merger of O_1 and O_2, a lookup made from n may end up on m as nodes may have updated their routing pointers/fingers to point to the other overlay. Though m is responsible for key k, it may not have yet received the data item for k as the merger has not completed at m. Thus, the data item for k will appear unavailable to node n. After the merge is complete, n will again be able to access k. A trivial solution to this problem is that when n learns that the key is currently unavailable, it retries the lookup after a while.

For some applications, availability may be critical. Thus, using retries after a while may not be a feasible solution. To provide higher availability guarantees when the overlays merger is in-progress, a node should store both its old routing pointers (that it originally had in its overlay) and new routing pointers (assigned by the merge process). Whenever a node receives a lookup, it routes the lookup through both, the old and new pointers. Thus, lookups will succeed even during the merge process. This approach, originally proposed by Datta [34], has some drawbacks. First, it increases the message complexity for lookups. Second, it may be difficult to keep track of which overlay a node belongs to, especially when more than one overlays are merged. Finally, since

knowledge of the completion of the merge process is not locally available to nodes, it is not known when to drop the old pointers. Nevertheless, we believe this to be valuable approach as it increases availability during merger.

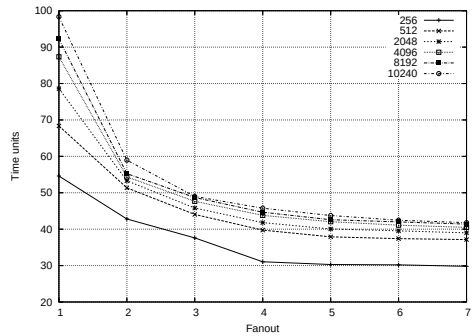
3.2.5 Evaluation

In this section, we evaluate the two algorithms from various aspects and in different scenarios. There are two measures of interest: *message complexity*, and *time complexity*. We differentiate between the *completion* and *termination* of the algorithm. By completion we mean the time when the rings have merged. By termination we mean the time when the algorithm terminates sending any more messages. Unless specified otherwise, message complexity is until termination, while time complexity is until completion.

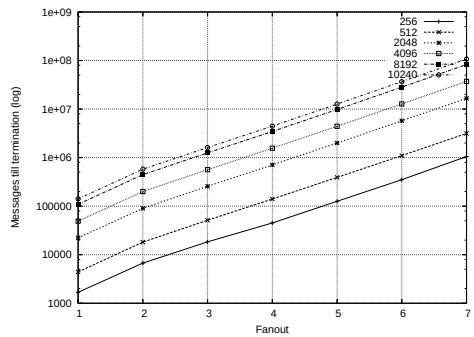
The evaluations are done in a stochastic discrete event simulator [144] in which we implemented Chord. The simulator uses an exponential distribution for the inter-arrival time between events (joins and failures). To make the simulations scale, the simulator is not packet-level. The time to send a message is an exponentially distributed random variable. The values in the graphs indicate averages of 20 runs with different random seeds.

We first evaluate the message and time complexity of the algorithms in a typical scenario where after merger, many nodes simultaneously detect alive nodes in their passive lists. Next, we evaluate the performance of the algorithm for a worst case scenario when only a single node detects the existence of another ring. The worst case scenario is similar to a case where an administrator wants to merge two overlays and triggers the ring unification algorithm on only a single node. Next, we assess the algorithms for a loopy ring. Thereafter, we evaluate the performance of the algorithms while node joins and failures are taking place during the ring merging process. Next, we compare our algorithm with a self-stabilizing algorithm. Finally, we evaluate the message complexity of the algorithms when a node falsely believes that it has detected another ring.

For the first experiment, the simulation scenario had the following structure. Initially nodes join and fail. After a certain number of nodes are part of the system, we insert a partition event, upon which the simulator divides the set of nodes into as many components as requested by the partition event, dividing the nodes randomly into the partitions but maintaining an approximate ratio specified. For our simulations, we create two partitions. A partition event is implemented using lot-



(a) Time complexity



(b) Message complexity

Figure 3.3: Evaluation for various network sizes and fanouts of a typical scenario where multiple nodes detect the merger and trigger the algorithm.

tery scheduling [163] to define the size of each partition. The simulator then drops all messages sent from nodes in one partition to nodes in another partition, thus simulating a network partition in the underlying network and therefore triggering the failure handling algorithms. Furthermore, node join and fail events are triggered in each partitioned component. Thereafter, a network merger event simply allows messages to reach other network components, triggering the detection of alive nodes in the passive lists, and hence starting the ring unification algorithms.

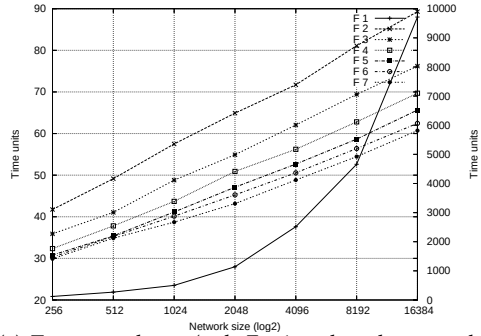
We simulated the simple ring unification algorithm and the gossip-based ring unification algorithm for partitions creating two components, and for fanout values from 1 to 7. For all the simulation graphs to follow, a fanout of 1 represents the simple ring unification algorithm.

Figure 3.3 shows the time and message complexity for a typical scenario where after a merger, multiple nodes detect the merger and thus start the ring-unification algorithm. The number of nodes detecting the merger depends on the scenario; in our simulations, it was 10–15% of the total nodes. The simple ring unification algorithm ($F = 1$) consumes minimum messages but takes maximum time when compared to different variations of the gossip-based ring unification algorithm. For higher values of F, the time complexity decreases while the message complexity increases. Increasing the fanout after a threshold value (around $3 - 4$ in this case) will not considerably decrease the time complexity, but will just generate many unnecessary messages.

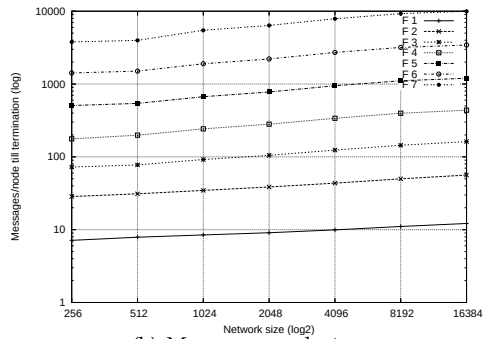
To proper understand the performance of the proposed algorithm, we generated scenarios where only one node would start the merger of the two rings. We randomly select, with uniform probability, the two nodes that are involved in the merger, i.e. the node p that detects the merger and the node that p detects from its passive list. Hence, the distance between them on the ring varies. For our experiments, each of the two rings had approximately half of the total number of nodes in the system before the merger. We choose the rate of checking *detqueue* to be every five time units and the rate of periodic stabilization (PS) to be every ten time units. The motivation for choosing a lower PS rate is to study the performance of the ring unification algorithm with minimum influence from PS.

We simulated ring unification for various network sizes of powers of 2 to study its scalability. Figure 3.4(a) shows the time complexity for varying network sizes. The x-axis is on a logarithmic scale, while the y-axis is linear. The graph for the gossip-based algorithms is linear, which suggests a $O(\log n)$ time complexity. In contrast, the simple ring unification graph (F=1) is exponential, indicating that it does not scale well, i.e. $\omega(\log n)$ time complexity. In Figure 3.4(b), we plot the number of ring unification messages sent by each node during the merger, i.e. the total number of messages induced by the algorithm until termination divided by the number of nodes. The linear graph on a log-log plot indicates a polynomial messages complexity. As expected, the number of messages per node grows slower for simple ring unification compared to gossip-based ring unification.

Figure 3.5 illustrates the tradeoff between time and message complexity. It shows that the goals of decreasing time and message complexity are conflicting. Thus, to decrease the number of messages, the time for completion will increase. Similarly, opting for convergence in lesser time will generate more messages. A suitable fanout value can be



(a) Time complexity (only F = 1 is plotted against the right y-axis)



(b) Message complexity

Figure 3.4: Evaluation for various network sizes and fanouts when only one node starts the merger.

used to adapt the ring unification algorithm according to the requirements and network infrastructure available.

For the rest of the evaluations, we use a worst case scenario where only a single node detects the merger.

Next, we assess the ability of our solution to converge to a *strongly stable* ring from a *loopy* state of two cycles. As defined by Liben-Nowell *et al.* [96], a Chord network is *weakly stable* if, for all nodes u, $(u.succ).pred = u$ and *strongly stable* if, in addition, for each node u, there is no node v such that $u < v < u.succ$. A loopy network is one which is weakly but not strongly stable. The scenario for the simulations was to create a loop of two cycles from one-fifth of the total number of nodes. Thereafter, we generated events of node joins for the remaining four-fifth nodes at an exponentially distributed inter-arrival rate. As in all experiments, the identifiers of the joining nodes were generated randomly with uniform

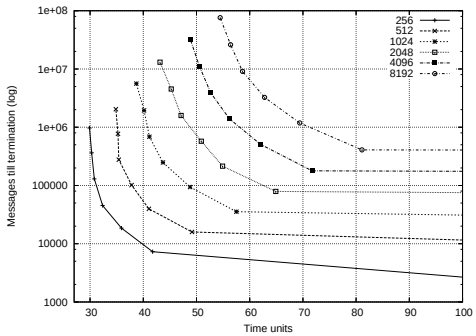
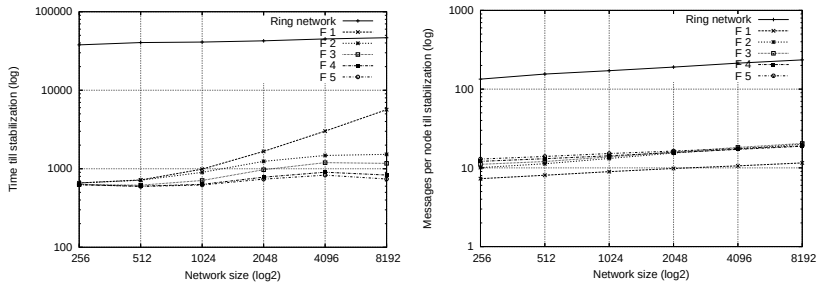


Figure 3.5: Tradeoff between time and message complexity.

probability. Thus, the nodes joined at different points in the loop. We then made one random node detect the loop by discovering a random node from the other cycle, triggering the ring unification algorithm. In all cases, for various network sizes and fanouts, the algorithm converged to a strongly stable ring, and the time and message complexity followed the same trend as for merging two overlays.

We evaluate ring unification under churn, *i.e.* when nodes join and fail during the merger. Since we are using a scenario where only one node detects the merger, with low probability, the algorithm may fail to merge the overlays (especially simple ring unification). The reason being intuitive: for simple unification, the two mlookups generated by the node detecting the merger while traveling through the network may fail as the node forwarding the mlookup may fail under churn. With higher values of F and in typical scenarios where multiple nodes detect the merger, the algorithm becomes more robust to churn as it creates multiple mlookups.

In our simulations, after a merge event, we generate join and failure events until the unification algorithm terminates and observe how often the overlays do not converge to a ring. We ran experiments with 200 different seeds for sizes ranging from 256 to 2048 nodes. We considered an execution successful if 95% of the nodes had correct successor pointers, as all successor pointers can not be correct while nodes are joining and failing. Thereafter, the remaining pointers are updated by Chord's periodic stabilization. For the 200 executions, we observed only 1 unsuccessful execution for network size 1024 and 2 unsuccessful executions for network size 2048. The unsuccessful executions happened only for simple ring unification, while executions with gossip based ring uni-



(a) Time complexity (b) Message complexity

Figure 3.6: Comparison of Ring Unification and SSRN [142].

fication were always successful. Even for the unsuccessful executions, given enough time, periodic stabilization updates the successor pointers to correct values.

Next, we compare our algorithm with a Self-Stabilizing Ring Network (SSRN) [142] protocol. The results of our simulations comparing time and message complexity for various network sizes are presented in Figure 3.6, depicting that ring unification consumes lesser time and messages compared to SSRN. The main reason for the better performance of our algorithm is that it has been designed specifically for merging rings. On the other hand, SSRN is a non-terminating algorithm that runs in the background like PS to find closer nodes. As evaluated previously, simple ring unification (fanout=1) does not scale well for time complexity (Figure 3.6(a)).

Finally, we evaluate the scenario where a node falsely detects a merger. In such cases, the algorithm should terminate quickly, without sending a lot of messages as they are unnecessary. Figure 3.7 shows the message complexity of the algorithm in case of a false detection. As can be seen, for lower fanout values, the message complexity is less. Even for higher fanouts, the number of messages generated are acceptable, thus showing that the algorithm is lean. We believe this to be important as overlays do not have perfect failure detectors, and hence can give rise to inaccurate suspicions.

In conclusion, our simulations show that a fanout value of 3-4 is good for a system with several thousand nodes, even with respect to churn and false-positives.

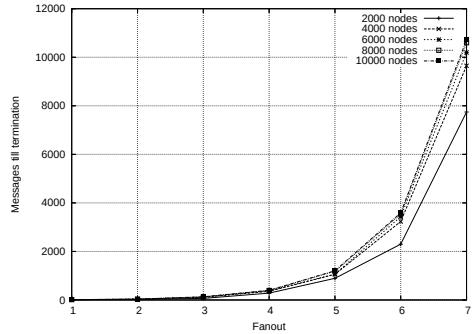


Figure 3.7: Evaluation of message complexity in case a node falsely detects a merger for various network sizes and fanouts.

3.2.6 Related Work

Much work has been done to study the effects of churn on a structured overlay network [102], showing how overlays can cope with massive node joins and failures, thus showing how overlays are resilient to partitions. Datta *et al.* [35] have presented the challenges of merging two overlays, claiming that ring-based networks cannot operate until the merger operation completes. In contrast, we show how unification can work under churn while the merger operation is not complete. In a followup work, Datta *et al.* [34] show how to merge two P-Grid [3] structured overlay networks. Their work differs from ours as P-Grid is a tree-based overlay, while we focus on ring-based overlays.

The problem of constructing a structured overlay from a random graph is, in some respects, similar to merging multiple structured overlays after a network merger, as the nodes may get randomly connected after a partition heals. Shaker *et al.* [142] have presented a ring-based algorithm for nodes in arbitrary state to converge into a directed ring topology. Their approach is different from ours, in that they provide a non-terminating algorithm which should be used to replace all join, leave, and failure handling of an existing overlay. Replacing the topology maintenance algorithms of an overlay may not always be feasible, as overlays may have intricate join and leave procedures to guarantee lookup consistency [101, 94, 47]. In contrast, our algorithm is a terminating algorithm that works as a plug-in for an already existing ring-based overlay.

Kunzmann *et al.* [79] have proposed methods to improve the robustness of structured overlays. They propose to use a bootstrapping

server to detect a merger by making the peer with the smallest identifier to send periodic messages to the bootstrap server. As soon as the bootstrap server receives messages from different peers, it will detect the existence of multiple rings. Thereafter, all the nodes have to be informed about the merger. While their approach has the advantage of having minimum false detections, it depends on a central bootstrap server. They lack a full algorithm and evaluation of how the merger will happen. Evaluation of the merge detection process and informing all peers about the detection is also missing.

Montresor *et al.* [109] show how Chord [153] can be created by a gossip-based protocol [68]. However, their algorithm depends on an underlying membership service like Cyclon [160], Scamp [45] or Newscast [69]. Thus the underlying membership service has to first cope with network mergers (a problem worth studying in its own right), where after T-Chord can form a Chord network. We believe one needs to investigate further how these protocols can be combined, and their epochs be synchronized, such that the topology provided by T-Chord is fed back to the overlay when it has converged. Though the general performance of T-Chord has been evaluated, it is not known how it performs in the presence of network mergers when combined with various underlying membership services.

As we show below, it might happen that an initially connected graph can be split into two separate components by the Chord [153] and SSRN [142] protocols. This scenario is a counter-proof of the claim that SSRN is self-stabilizing. Consider a network which consists of two perfect rings, yet the nodes have fingers pointing to nodes in the other ring. This can easily happen in case of unreliable failure detectors [23] or networks partitions. Normally, the PS rate is higher than fixing fingers, thus due to a temporary partition, it might happen that nodes update their successor pointers, yet before they fix their fingers, the partition heals. In such a scenario, SSRN splits the connected graph into two separate partitions, thus creating a partition of the overlay, while the underlay remains connected. An example of such a scenario is shown in Figure 3.8, where the filled circles are nodes that are part of one ring and the empty circles are nodes that are part of the other ring. Each node has one finger pointing to a node in the other ring. The fix-finger algorithm in Chord updates the fingers by making lookups. In this case, a lookup will always return a node in the same ring as the one making the lookup. Consequently, the finger pointing to the other ring will be lost. Similarly, the pointer jumping algorithm used by SSRN to update its fingers will also drop the finger pointing to a node in the other ring.

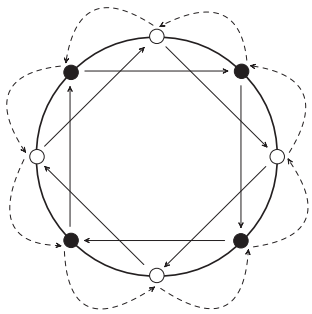


Figure 3.8: A case where Chord and the Ring Network protocol would break a connected graph into two components. Lines represent successor pointers while dashed lines represent a finger.

On the contrary, the ring-unification algorithm proposed in this section will fix such a graph and converge it to a single ring.

Some overlays employ the ring based identifier space, which they mix with a prefix-based tree [121]. For example in Pastry [127], a responsible node for an identifier is the node with numerically closest identifier and the lookups are forwarded to nodes sharing the longest prefix with the identifier being looked up. Our algorithm can be modified for use by such overlays by replacing the closestpreceedingnode-procedure with the equivalent for the employed overlay. The trymerge-procedure does not have to be changed since updating the predecessor and successor is similar to recording nodes with identifiers closest to a node.

3.3 Recircle: Bootstrapping, Maintenance, and Mergers

In this section, we present an overlay algorithm, called *ReCircle*, that is capable of *(i)* bootstrapping an overlay, *(ii)* maintaining the overlay under churn, and *(iii)* handling underlying network partitions and mergers. The algorithm builds and maintains a structured overlay with a uni-directional ring geometry. It allows a system designer to trade-off between bandwidth consumption and time taken for bootstrapping and merging overlays. During normal operation, i.e. after bootstrapping and without underlying network partitions and mergers, the algorithm behaves similar to a general overlay maintenance algorithm without any overhead. We designed ReCircle using principles taken from Ring-Unification and Periodic Statbilization.

Motivation Efficient bootstrapping - creating a populated structured overlay network from scratch - is a challenge that has not been addressed by overlays. Overlays are limited in the rate at which new nodes can join the overlay [95], hence the time duration needed to create an overlay of a large size may be long. This approach has two drawbacks. First, the system users may have to wait for a long duration, depending on the size of the overlay, before they can use the overlay. This is undesirable for example when resources are allocated for limited duration of time, or when an overlay has to be created in an ad hoc or temporary setting. Second, limiting the rate of joins may be complicated or require central coordination, which defies the ideology of structured overlay networks as they are decentralized peer-to-peer systems. Hence, given the goal of structured overlays to be self-managing and self-organizing, we believe that they should be able to bootstrap efficiently without constraints on the size of the overlay or the rate of joins.

As argued in Chapter 3.1, network partitions are a fact of life, and hence, overlays should be able to cope with them. We strongly believe that if structured overlay networks are to realize their goal of being scalable, fault-tolerant, self-managing and self-organizing, they should *inherently* be able to bootstrap efficiently, and handle network partitions and mergers other than only being able to deal with moderate rates of churn. The goal of ReCircle is to provide such an overlay maintenance algorithm.

Solution

Bootstrapping, network partitions and mergers, and flash crowds represent extreme rates of churn. Bootstrapping and overlay mergers are similar to a large number of nodes joining the overlay simultaneously, where as network partitions are akin of massive failures. Flash crowds can be either huge number of nodes joining or leaving the overlay. Periodic stabilization (PS) can handle massive failures [96] as long as no node looses all its successor-list. Hence, if no node has its successor-list partitioned away, PS can handle network partitions, making each component of the partition eventually form its own ring. Furthermore, overlays cannot intrinsically bootstrap efficiently, handle flash crowds of joins, or deal with overlay mergers. To handle all these cases, we propose ReCircle, an overlay maintenance algorithm that runs periodically for normal overlay maintenance, and reacts to extreme events and starts sending messages other than the periodic messages. Periodic messages are exchanged between a node, its successor and predecessor to maintain the geometry in the node's immediate vicinity only, while the re-

active messages can navigate further in the identifier space, similar to Ring-Unification. These reactive messages remedy the anomalies in the geometry and the overlay converges to a ring. Once the overlay converges, the reactive messages die out and the algorithm returns to act as a normal periodic maintenance algorithm.

Our methodology is different from overlay maintenance algorithms such as Chord's periodic stabilization in two aspects. First, ReCircle is reactive to extreme events, while Chord is always periodic. Being reactive is desirable for extreme events since such events invalidate several pointers simultaneously. Second, in Chord, a node periodically attempts to fix any possible anomalies in the geometry only with its immediate successor. On the other hand, as extreme events may quickly make the immediate neighbourhood of a node on the ring outdated, ReCircle is able to traverse farther away, using an operation similar to an overlay's lookup and Ring-Unification's mlookup.

Our solution is given as Algorithm 5. Periodically, every δ time units (line 1), each node n attempts to set it's *succ* to a node clockwise closer to n than n's current successor. n accomplishes this by retrieving its successor's *pred* pointer, and updates *succ* if it finds a closer successor.

Each node maintains a *queue*, which contains a list of node identifiers that represent possible (problematic) areas on the identifier space that violate the geometry of the overlay and can be fixed. These areas can arise, for example, due to churn, bootstrapping, and flash crowds. If the queue is nonempty at any node, it implies that the overlay may not be in a converged state. Later in this section, we discuss all cases in which node identifiers should be added to the queue.

ReCircle uses an event called mlookup(id) (similar to Section 3.2.1) for fixing a possible problematic area *id* on the identifier space. An mlookup(id) does the following. First, it performs a greedy routing, similar to a Chord lookup, to the problem area defined by the identifier *id*. Once it routes to *id*, it fixes the geometry there by triggering the same afore-mentioned mechanism that is periodically carried out every δ time units. The mlookup then continues to fix the ring in the clockwise direction (Figure 3.10). Second, an mlookup spreads the fixing process by generating new mlookups for random identifiers on the ring; hence triggering the fixing mechanism at random places on the identifier space. This is accomplished by enqueuing *id* into random nodes from the nodes routing table (lines 25–26). This is shown in Figure 3.9. Third, as an optimization, an mlookup attempts to optimistically fix any wrong successor and predecessor pointers while routing by calling the Update method (line 35).

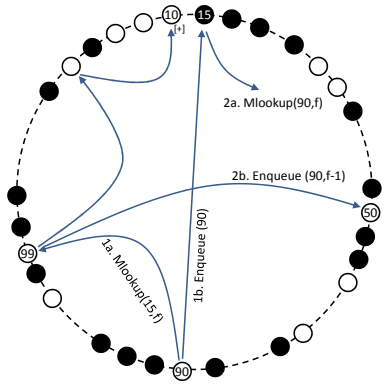


Figure 3.9: White nodes belong to an overlay $O1$, while black nodes belong to another overlay, $O2$. The merger starts when 15 is added to the queue of 90. (1a) 90 makes an mlookup(15) to fix the ring geometry around identifier 15, and also asks 15 (1b) to make an mlookup for 90, which will result in fixing the ring around 90 (2a). While routing the mlookup(15), 99 shares the merger information with a random node, 50, from its routing table (2b). This will eventually result in an mlookup(50) from 15 that will fix the ring around 50 (not shown in figure). Details of mlookup(15) ending at 10, denoted as [+], are shown in Figure 3.10.

Periodically, after every γ time units (line 15), each node tries to fix the geometry of the overlay by generating mlookups to identifiers in its queue. Furthermore, whenever p makes an mlookup(q), then q also makes an mlookup(p).

ReCircle provides knobs to tradeoff bandwidth consumption and the time taken to converge to a ring geometry. This tradeoff can be achieved by controlling the amount and rate of spreading the fixing procedure. The number of times the fixing procedure is spread is equivalent to the number of new mlookups generated. As mentioned earlier, new mlookups are generated while routing an mlookup (lines 25–26). Here, we employ a *fanout* parameter f that controls how many new mlookups are generated (line 23). Higher values of the fanout will result in more concurrent mlookups, hence consuming more bandwidth but converging in lesser time. Similarly, the rate of spreading the fixing procedure is equivalent to the rate at which mlookups are started. Since new mlookups are started periodically by dequeuing, this rate can be controlled via the time period γ, and the number of mlookups generated in each period, denoted as Mlkups_Per_Period (line 16).

Algorithm 5 ReCircle

```
1: every δ time units at p
2:     sendto succ : GetPred⟨succ⟩
3: end event

4: receipt of GetPred⟨psucc⟩ from m at n
5:     sendto m : GetPredRes⟨pred, sl⟩
6:     if psucc ≠ n then
7:         queue.enqueue(⟨psucc, f⟩)
8:     end if
9:     UPDATE(m)
10: end event

11: receipt of GetPredRes⟨succp, succsl⟩ from m at n
12:     UPDATE(succp)
13:     UPDATESUCCESSORLIST(succsl)
14: end event

15: every γ time units and queue ≠ ∅ at p
16:     for i ← 1:Mlkups_Per_Period and queue ≠ ∅ do
17:         ⟨q, F⟩ := queue.dequeue()
18:         sendto p : MLookup⟨q, F⟩
19:         sendto q : MLookup⟨p, F⟩
20:     end for
21: end event

22: receipt of MLookup⟨id, F⟩ from m at n
23:     if F > 1 then
24:         F := F − 1
25:         r := randomnodeinRT()
26:         at r : queue.enqueue(⟨id, F⟩)
27:     end if
28:     if id ≠ n and id ≠ succ then
29:         if id ∈ (n, succ) then
30:             sendto id : GetPred⟨succ⟩
31:         else
32:             sendto closestpreceding(id) : MLookup⟨id, F⟩
33:         end if
34:     end if
35:     UPDATE(id)
36: end event

37: procedure Update(candidate) at n
38:     if candidate ∈ (n, succ) then
39:         succ := candidate
40:     else if pred = nil or candidate ∈ (pred, n) then
41:         pred := candidate
42:     end if
43: end procedure
```

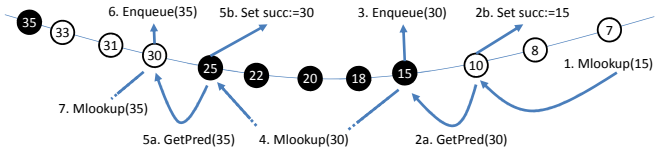


Figure 3.10: White nodes belong to an overlay $O1$, while black nodes belong to another overlay, $O2$. The figure depicts how new mlookups are generated when an mlookup(15) terminates at 10; where $10 \in O1$ and $15 \in O2$. Here, the new mlookups enable the algorithm to continue merging the ring clock-wise.

3.3.1 Merging multiple overlays

Two independent overlays can be merged into a single overlay using Algorithm 5. The merger can be triggered via connecting the overlays by adding the identifier of any node from one overlay to the queue of any node from the other overlay[2] either by the passive lists mechanism (Section 3.1.1), or an administrator. An mlookup will be generated for the node in the queue. As noted earlier, an mlookup(m) first routes to the problematic area m, terminating at a node n such that $m \in [n, n.succ]$. Then, the geometry is fixed by setting $n.succ := m$, and the two overlays are merged on the identifier space around identifier m. The merger process is continued by issuing new mlookups.

Consider Figure 3.10, where $n = 10$ and $m = 15$. An mlookup, to propagate the merger, is needed between 10 and $10.succ = 30$ because a merger between overlays $O1$ and $O2$ can result in several nodes from $O2$ to be placed between a node 10 in $O1$ and 10's successor. 10 accomplishes this propagation by asking 15 (step 2a) to enqueue 30 (line 7) as it represents a problematic area. Such a mechanism enables ReCircle to continue merging the overlays clock-wise. Furthermore, as new mlookups are generated for random identifiers while routing an mlookup, the overlays concurrently merge clock-wise starting at random positions in the identifier space and eventually, converge into one overlay (Fig. 3.9). Note that ideally, the new mlookups are generated such that the source and destination nodes belong to different overlays.

[2]The higher the number of connections between the two overlays, the faster the overlay will converge.

3.3.2 Bootstrapping

The ideas for merging two overlays apply to merging more than two overlays as well; an extreme case of which is bootstrapping where each node can be considered an overlay in itself. Bootstrapping is achieved by creating a structured overlay from a random connected overlay, where each node has some random nodes as neighbours. In our algorithm, each node can be considered an independent structured overlay of size one by pointing to itself as its successor and predecessor. To start bootstrapping, each node adds its neighbours to its *queue*. The algorithm then triggers the merger mechanism by generating mlookups to nodes in the queue, resulting in a single converged overlay.

3.3.3 Termination

An important requirement for a unified algorithm is that under normal scenarios (i.e. no churn), the maintenance cost should be low, for instance, similar to Chord's periodic stabilization. To achieve this, we designed the algorithm to be reactive such that it starts generating more messages than the periodic maintenance mechanism to handle rare events such as bootstrapping or network partitions and mergers. Once such events are catered and the overlay converges, the algorithm stops sending extra messages and the number of messages drops to the only periodic maintenance messages. This property is achieved by not generating new mlookups when the possible problematic area is already fixed (lines 28 and 6). When the overlay is converged, there will not be any problematic areas and hence, the queues on all nodes will eventually be emptied and no new mlookups will be started.

3.3.4 Evaluation

In this section, we evaluate ReCircle by both simulations and experiments on Planetlab[3]. We implemented the algorithm in Kompics [10] and for simulations, we used the King latencies [59] for network delays. The focus of the evaluation is on overlay mergers and bootstrapping as normal scenarios are handled similar to Chord. The two main metrics used are bandwidth consumption and time taken for convergence. The simulations were repeated with 20 different random seeds, and we plot average and 95% confidence intervals in our graphs.

[3]http://www.planet-lab.org

Parameter		Values
Fanout	f	1 – 5
mlookups per period	m	1 – 5, ∞
Queue interval	γ	1, 2 (secs)
Periodic maintenance interval	δ	10, 30, 60 (secs)

Table 3.1: Range of parameter values used for simulations.

Same size networks merge

We first consider the performance of the algorithm when two overlays of same size merge. As the simulation scenario, we created two separate overlays of the same size, and then started the merger algorithm by creating one link between the overlays.

Algorithm 5 uses four parameters:

- *f:* The fanout *f* used to control the spread of mlookups (line 7).
- *m:* The number of mlookups generated in each period γ, shown as Mlkups_Per_Period on line 16.
- γ*:* The interval after which mlookups are generated to identifiers stored in the queue (line 15).
- δ*:* The interval after which a node performs periodic stabilization (line 1).

To study the affect of all four parameters, we employed a performance-vs-cost model [93] where we used ranges of values for each parameter. The ranges for parameter values we chose for evaluation are shown in Table 3.1. We used higher values of δ, compared to γ, in line with Li's study [91] on comparing range of values of periodic interval for maintenance in various overlays. In Table 3.1, $m = \infty$ means that the mlookups are not queued but are instead generated instantly. Each combination of the parameter values was simulated for 20 different random number generation seeds. For each simulation, the combination of parameters had some cost and performance associated with it. For our work, the cost of the algorithm is the *bandwidth* used per peer during the merge process, and the performance is the *time* taken by the algorithm to converge the overlays into one overlay.

Figure 3.11 shows the results of various combinations of the parameters for a total network size of 2048. Each dot in the graph represents the result of a single experiment for a parameter combination. As is evident from the figure, when the cost is more (higher bandwidth),

the performance is better (lower time to convergence). Similarly, less cost (lower bandwidth) results in lower performance (high convergence time). Furthermore, there is a point after which more cost does not increase performance. Similarly, there is a limit to the minimum cost.

Further analysis of the performance-vs-cost experiment (Fig 3.11) shows that δ does not influence the results much. Similarly, increasing γ from 1 second to 2 seconds does not help much either. Hence for the next evaluations, we use $\delta = 60$ seconds and $\gamma = 1$ second.

Affect of Fanout (f) and mlookups per period (m)

Next, we discuss the affect of f and m on the cost and performance of the algorithm. Figure 3.12 shows the convergence time, while Figure 3.13 shows the bandwidth consumption, for different values of f and m. For $f = 1$, the convergence time is high, yet ReCircle consumes minimum bandwidth. This is an expected behaviour as concurrent mlookups are not generated when $f = 1$ at line 25 and the merge process continues linearly. Similarly, as we increase the value of f, the convergence time drops slower, while the bandwidth increases exponentially. This trend applies to all simulated values of m, which implies that after a certain value of f, increasing f will only increase cost without significant improvement in performance.

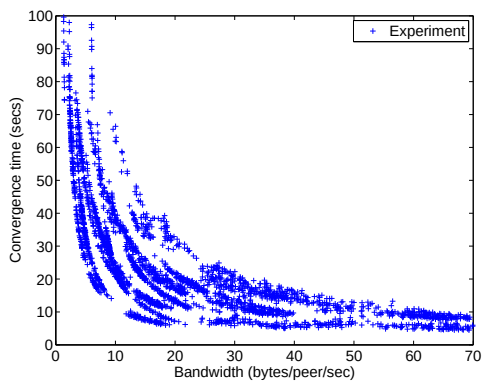


Figure 3.11: A performance vs cost comparison when two networks, each of size 1024, merge.

Figure 3.14 and 3.15 plot the performance and cost respectively for various values of m. The bandwidth consumption increases logarithmically with m, while time to convergence drops slowly.

An important aspect of the algorithm is that in case there is no churn, ReCircle only sends the periodic maintenance messages. As soon as a rare event that results in churn occurs, such as merger of multiple overlays, the algorithm reacts to it by consuming more bandwidth. Once the overlay converges, the overhead messages die out and the bandwidth consumption drops back. This is shown in Figure 3.16, where two overlays are merged after 10 seconds. The Y-axis denotes the bandwidth consumed per peer in every 200 milliseconds. As evident from the figure, bandwidth consumption increases to merge the overlays. Once the overlays converge into a single overlay, the bandwidth consumption reduces to the level of before the merger.

Set successor calls during merger

Distributed applications build on top of a structured overlay network assign responsibilities to participating nodes based on the region of the identifier space between a node, and its successor and predecessor in the overlay. A change in the successor or predecessor pointers of a node n re-assigns responsibilities between nodes in n's vicinity, which requires action on behalf of the application. For instance, in DHTs built on overlays e.g. Cassandra, a node is responsible for storing all data items with keys between it's identifier and it's immediate neighbour's identifiers. Here, whenever a successor pointer changes, responsibilities are re-defined and data has to be transferred from one node to another. Hence, it is desirable to have a minimum number of unnecessary

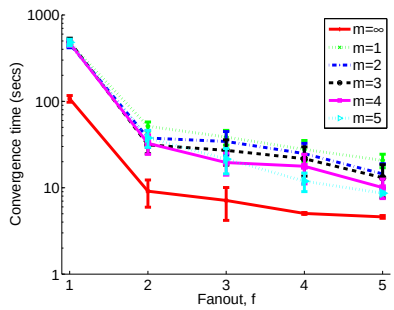


Figure 3.12: Convergence time for various values of f, where $n = 2048$, $delta = 60$ secs, and $\gamma = 1$ sec.

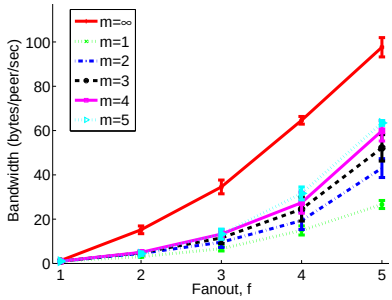


Figure 3.13: Bandwidth consumption for various values of f, where $n = 2048$, $delta = 60$ secs, and $\gamma = 1$ sec.

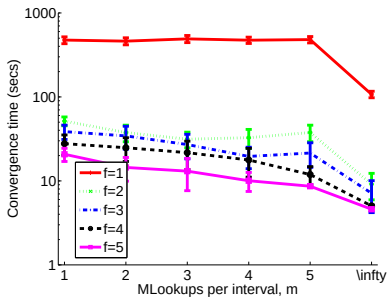


Figure 3.14: Convergence time for various values of m.

calls to set the successor of a node, for instance, during merging multiple overlays to avoid unneeded data transfers. In this experiment, we merged two overlays and measured the number of set successor calls, s, and compared it to the number of incorrect successors w at the point

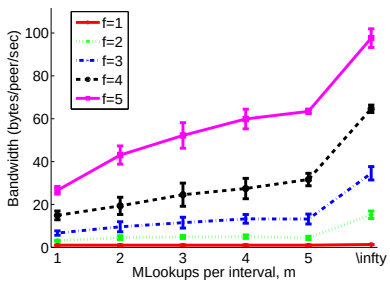


Figure 3.15: Bandwidth consumption for various values of m.

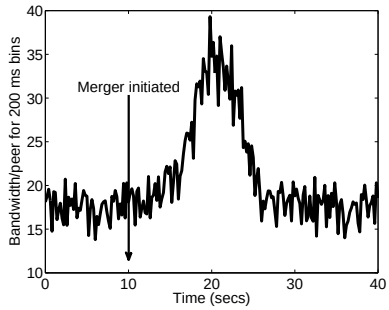


Figure 3.16: Bandwidth consumption for 200 milliseconds bins, showing termination of reactive messages after convergence.

the overlays started to merge. Ideally, s should be equal to w, but is difficult to achieve due of decentralization. Figure 3.17 shows the ratio $\frac{s}{w}$ for a range of values of f (1–5) and m (1–5, and ∞). The graph shows that $f = 1$ has the minimum ratio, and hence would result in minimum data transfer. This shows that if an overlay stores huge data items under keys, the overall time (time for correcting routing pointers and moving data items to new responsible nodes) for $f = 1$ might be lesser than for larger values of f. In the light of this experiment, when higher values of f are used, instead of immediately transferring data when the responsibility of a node changes, a periodic or delayed data exchange mechanism should be used to transfer data among nodes. Using such a technique will avoid transferring data unnecessarily when the merger is under progress.

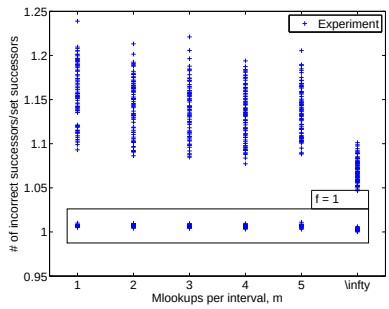


Figure 3.17: Ratio, during the merger process, of the number of times successor is set versus the number of incorrect successors when the merger started.

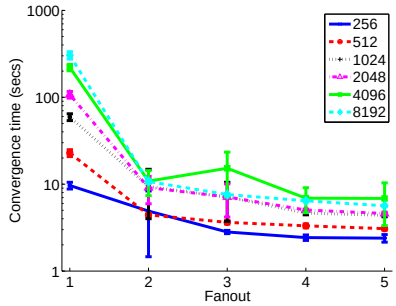


Figure 3.18: Convergence time for various network sizes, where $m = \infty$, $t = 60$ secs, and $r = 1$ sec.

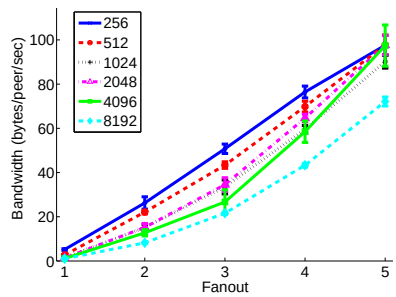


Figure 3.19: Bandwidth consumption for various network sizes, where $m = \infty$, $t = 60$ secs, and $r = 1$ sec.

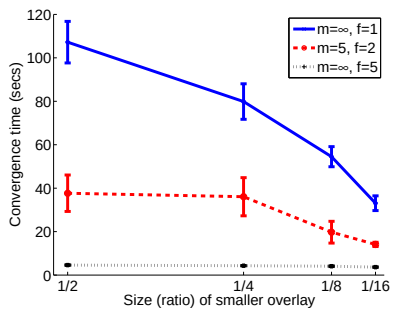


Figure 3.20: Convergence time when overlays of different size merge.

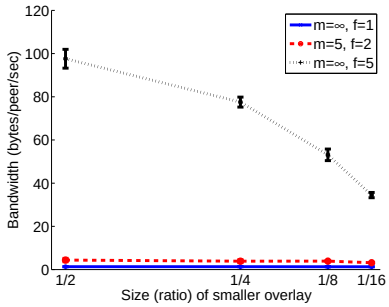


Figure 3.21: Bandwidth consumption when overlays of different size merge.

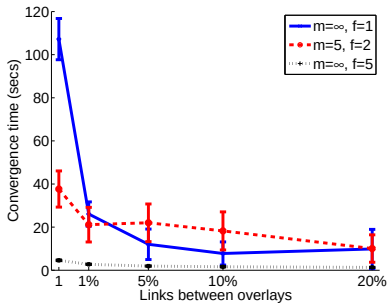


Figure 3.22: Convergence time when multiple links trigger the merge process.

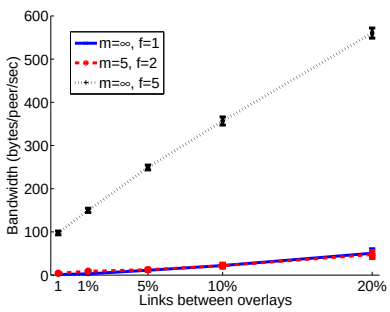


Figure 3.23: Bandwidth consumption when multiple links trigger the merge process.

Varying network size

Next, we studied the effect of f on the algorithm for different network sizes, while using $m = \infty$. Figure 3.18 and 3.19 show the convergence time and bandwidth consumption for various network sizes, depicting that the trend remains the same. The convergence time for $f = 1$ is high, while for higher values of f, it drops to a certain level after which further increasing f does not reduce convergence time (Fig. 3.18). On the other hand, the cost only grows logarithmically with increase in f even for larger network sizes (Fig. 3.19).

Different size networks merge

In this section, we evaluate the cost and performance of the algorithm when overlays of different sizes merge, which is a common scenario as network partitions are usually of unequal sizes [108]. We expect that the cost should be proportional, and the performance should be inversely proportional, to the size of the smaller network. The reason is that at the start of the merger, when using uniformly random identifiers, the number of wrong successor pointers depends on the size of the smaller network. For instance, for a total network size of 100 nodes, the cost of merging two overlays of sizes 80 and 20 should be lesser than merging overlays of sizes 60 and 40.

In our simulations, we created two overlays of different sizes, and then started the merger by creating a single link between the overlays. Figure 3.20 and 3.21 show the results for a total network size of 2048. The x-axis depicts the ratio of the smaller network out of the total network size. We plot results for three combinations of m and f: $m = 2, f = 1$ (high convergence time, low bandwidth consumption), $m = 4, f = 3$ (medium convergence time and bandwidth consumption), and $m = \infty, f = 5$ (low convergence time, high bandwidth consumption). The results confirm that when a smaller network merges into a larger network, the algorithm consumes resources relative to the smaller network. Hence, as the size of the smaller network decreases, the algorithm requires lesser time for convergence and bandwidth.

Multiple links start merger

Next, we evaluate a scenario where the merger between two overlays is triggered by creating multiple links between the two overlays instead of a single link. This can happen when multiple nodes detect a network partition and merger. In such a scenario, the merger will be started simultaneously at multiple positions on the identifier space. Intuitively,

for higher number of inter-overlay links, the overlays should converge faster while consuming higher bandwidth because the algorithm reacts to the merger concurrently for all the inter-overlay links. This was confirmed in our simulations, as shown in Figures 3.22 and 3.23. The x-axis represents the number of links created between the two overlays for triggering the merger. The percentage on the x-axis is out of the total network size of 2048. The figures depict that, while multiple links can reduce the time to convergence, it results in higher bandwidth consumption. Higher percentages of links make $f = 1$ behave like $f > 1$ since the merger happens concurrently at different areas on the identifier space even for $f = 1$.

Bootstrapping

As discussed in Section 3.3.2, Algorithm 5 can be used for bootstrapping an overlay by considering each node as an overlay of size one and connecting the nodes randomly. In this section, we evaluate the performance of our solution for bootstrapping an overlay of size 2048. We create a random Erdős-Rényi graph $G(n, p)$, where $n = 2048$ and $p = \frac{\ln(n)}{n}$, and ensure that the graph is connected. The graph dictates the layout of the initial overlay that has to be bootstrapped into a converged structured overlay. Each node sets itself as its successor and predecessor, and the bootstrapping process is triggered by making each node add its neighbours to its queue.

We compare our solution to T-Man [72], a well-known gossip-based approach for creating arbitrary structured overlays from a random graph. In T-Man, the last few pointers take time to converge [68], hence, we measure statistics until 99% of the successor pointers are converged. To perform an extensive comparison for bootstrapping between T-Man and our solution, and not to depend on parameter tuning, we again employ a performance-vs-cost model. We use a range of parameter values and repeat each experiment for different seeds. For T-Man, we use the values specified in Table 3.2. The results are plotted in Figure 3.24, which shows that for the same cost (bandwidth consumption), both algorithms have similar performance in terms of convergence time. A disadvantage of using a specialized bootstrapping algorithm, such as T-Man, is that it requires handing off the bootstrapped overlay to the maintenance protocol which is non-trivial. In comparison, ReCircle does not require such a hand off as it embeds the overlay maintenance logic as well. We discuss such differences and benefits of using ReCircle in detail in Section 3.3.5.

Parameter	Values
Omega, ω	1 – 5
Message size	10, 20, 30, 40, 50
Gossip time period	0.2, 0.5, 1, 1.5, 2 (seconds)
Storage size	2048

Table 3.2: Range of parameter values used for T-Man [68].

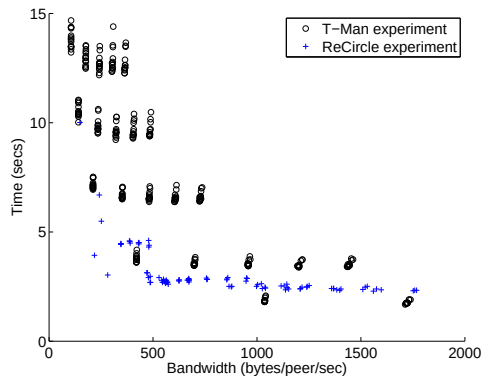


Figure 3.24: A comparison with T-Man [68] for creating a ring-structured overlay from a random Erdős-Rényi graph for a network size of 2048

PlanetLab

Next, we evaluate the solution for merging multiple overlays and bootstrapping on a real environment by running experiments on PlanetLab. Due to limited number of physical machines available on PlanetLab, we ran 5 nodes on each machine. We used a single server to gather statistics about how many nodes have a correct successor pointer. Whenever a node updated its successor, it sent a message to the statistics server with the new value of its successor. We compute the fraction of correct successor pointers overtime on the statistics server as it has the identifiers of all the nodes in the system and the values of each node's current successor.

In the first set of experiments, we evaluated the performance of merging two equal-sized overlays. We created two independent overlays and triggered the merger process by creating a link between the overlays. This was done by adding a random node from one overlay to the queue of a random node in the other overlay. The results for network sizes of

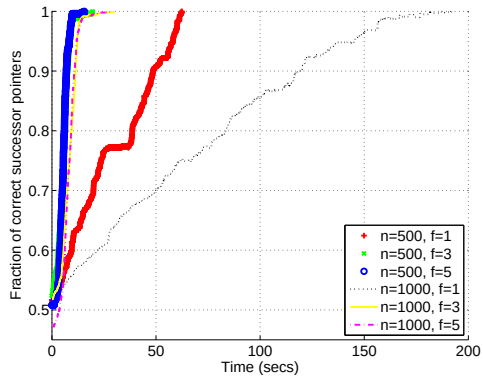


Figure 3.25: Evaluation on PlanetLab for merging two overlays.

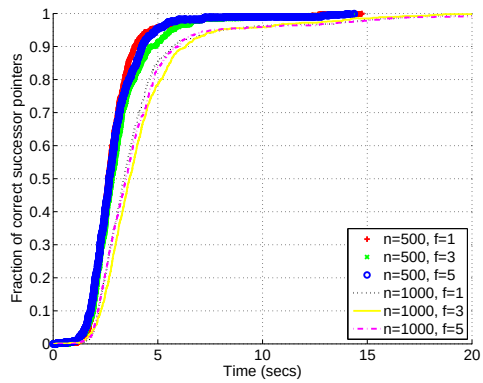


Figure 3.26: Evaluation on PlanetLab for bootstrapping an overlay.

500 and 1000 are shown in Figure 3.25. Analogous to the simulation results, the convergence rate for $f = 1$ is slower compared to higher values of fanout. Furthermore, with $f = 3$ and 5, most of the successor pointers converge within 10 to 15 seconds.

Finally, we evaluated the performance of bootstrapping an overlay on PlanetLab. We started the nodes as single node overlays by making each node point to itself as its successor and predecessor. As in the simulations, in this experiment we created an Erdős-Rényi graph at the statistics server using the identifiers of the nodes. Then, the statistics server initiated the bootstrapping process by sending messages to all nodes containing their neighbours as dictated by the generated graph. On receiving such a message, each node added the neighbours to their

queue, which triggered the bootstrapping mechanism. Figure 3.26 depicts the results for this experiment, which shows that an overlay of size 1000 can be created within 10 to 15 seconds. In the figure, $f = 1$ performs the same as higher values of f because all nodes are already participating in the merger process.

3.3.5 Related work

A variety of overlay maintenance algorithms have been proposed over the years [153, 62, 104, 91], and much work has been done to show their resilience to handle churn [96, 102]. These systems can cope with massive failures, thus being able to cope with network partitions as long as a node doesn't loose all its successor-list. Yet, these systems are not intrinsically designed for fast bootstrapping, and cannot merge multiple overlays. In this chapter, we show an overlay algorithm that can deal with such extreme events, while being able to perform periodic maintenance like any overlay algorithm. We believe that the underlying principles can be used in other overlays as well.

Bootstrapping a structured overlay is done by constructing a geometry, such as a ring in Chord, from a randomly connected overlay. Shaker *et al.* [142] have presented an algorithm, called Ring Network (RN), for nodes in arbitrary state to converge into a ring topology. While their algorithm can be used for overlay maintenance as well, it cannot converge from certain scenarios [136]. Furthermore, since their algorithm is not reactive to extreme events, it suffers from the same problems as other overlays where the time for convergence when two overlays merge is huge [136].

Montressor *et al.* show how any topology [68], such as a ring [109], can be created from a randomly connected overlay using a gossip-based protocol. However, in their algorithm, it is difficult to detect when the overlay has converged due to decentralization, and thus it depends on heuristics to detect termination. Hence, we believe that further investigation is required to study how these algorithms can be synchronized such that once the topology is built, it can be handed over to the overlay maintenance protocol. On the other hand, ReCircle does not require any such handover.

Datta *et al.* [35, 34] show how to merge multiple P-Grid [3] overlays. P-Grid is a tree-based overlay; in contrast, we focus on ring-based overlays. Furthermore, Similar to Ring-Unification, their algorithm is triggered for performing the merger, and then terminates after convergence, thus giving the control back to the overlay maintenance protocols. This can lead to two problems. First, the implications of such

a terminating algorithm on the overlay maintenance algorithm is not well studied. Coupled with a separate bootstrapping protocol further complicates the interaction between the algorithms. Second, a system developer has to implement and maintain separate mechanisms to address each problem, which can lead to unnecessary complexities. ReCircle provides a single algorithm that covers all such scenarios.

3.4 Discussion

Structured overlay networks are designed for dynamic environments and touted to be scalable, fault-tolerant and self-organizing. Therefore, apart from dealing with normal churn rates, we argue that they should intrinsically be able to handle rare but extreme events such as bootstraping, flash crowds, and network partitions and mergers. We have presented a mechanism to detect when a network partition seizes.

We have presented various algorithms to merge multiple ring-based overlays into one overlay. We have proposed, Ring-Unification, an overlay merger algorithm that terminates once the separate overlays are dissolved into one. We have also proposed a unified non-terminating algorithm ReCircle. Under normal execution, ReCircle algorithm exchanges messages periodically like any other overlay maintenance protocol. On the other hand, ReCircle is reactive to extreme events, such as bootstrapping and merger, so that it can converge faster when such events occur. We have designed Ring-Unification and ReCircle such that they provide tunable knobs to tradeoff between cost (bandwidth consumption) and performance (time to convergence) while handling extreme scenarios.

CHAPTER 4

Network Size Estimation

In this chapter, we present a *gossip-based* [119] *aggregation-style* [70] network size estimation algorithm for ring-based structured overlay networks. We demonstrate how our solution is robust to node failures compared to existing aggregation solutions.

Motivation Structured overlay networks are highly scalable and the number of nodes in the system fluctuates all the time due to churn. The network size is, however, a global variable which is not accessible to individual nodes in the system as they only know a subset of the other nodes. This information is, nevertheless, of great importance to many structured P2P systems, as it can be used to tune the rates at which the topology is maintained. Moreover, an estimate of the network size can be used in structured overlays for load-balancing purposes [55], deciding successor-lists size for resilience to churn [95], choosing a level to determine outgoing links [103], and for designing algorithms that adapt their actions depending on the system size [15].

Since knowledge of the size of the overlay is a core requirement for many systems, estimating the size in a decentralized manner is a challenge taken up by recent research activities. Out of these, gossip-based aggregation algorithms [71], though having higher overhead, provide the best accuracy [107]. Consequently, we focus on gossip-based aggregation algorithms in our work. While aggregation algorithms can be used to calculate different aggregates, e.g. average, maximum, minimum, variance etc., our focus is on counting the number of nodes in the system.

Although Aggregation [71] provides accurate estimates, it suffers from a few problems. First, Aggregation is highly sensitive to the *overlay topology* that it is used with. Convergence of the estimate to the real network size is slow for non-random topologies. On the contrary, the majority of structured P2P overlays have non-random topologies. Thus, it is not viable to directly use Aggregation in these systems. Second, Aggregation works in rounds, and the estimate is considered converged after a predefined number of rounds. As we discuss in section 4.3.1, this can be problematic. Finally, Aggregation is highly sensitive to node failures.

In this chapter, we propose a gossip algorithm based on Aggregation to be executed continuously on every node to estimate the total number of nodes in the system. The algorithm is aimed to work on structured overlay networks. It is robust to failures and adaptive to the network delays in the system.

4.1 Gossip-based Aggregation

In this section, we describe the original Aggregation algorithm suggested by Jelasity et. al. [71]. The algorithm is based on push-pull gossiping, shown in Algorithm 6.

Each node p has a local state, denoted as s_p in Algorithm 6. A node has two threads, an *active* thread, and a *passive* thread. The active thread is responsible for initiating push requests to other nodes, essentially starting an exchange of local states between two nodes. The passive thread acts as the pull mechanism; replying with local state to any push requests from other nodes.

The method GetRandomNeighbour returns a uniformly random sampled node over the entire set of nodes. Such a continuous and updated list of random nodes can be provided by an underlying peer sampling service, such as Newscast [69] or Cyclon [162]. The method Update computes a new local state based on the node p's current local state s_p and the remote node's state s_q. The goal for such a method is to reduce the variance in the states, a mechanism known as *anti-entropy* [119].

The time interval δ after which the active thread initiates an exchange is called a *cycle*. Given that all nodes use the same value of δ, each node roughly participates in two exchanges in each cycle, one as an initiator and the other as a recipient of an exchange request. Thus, the total number of exchanges in a cycle are roughly equal to $2 \times n$, where n is the network size.

Algorithm 6 Push-pull gossip in Aggregation [71]

1: **every** δ **time units at** p
2: q := GetRandomNeighbour()
3: **sendto** q : ExchangeState$\langle s_p \rangle$
4: $s_q \leftarrow$ ReceiveState(q) ▷ wait for q's state
5: $s_q :=$ Update(s_p, s_q)
6: **end event**

(a) Active thread

7: **receipt of** ExchangeState$\langle s_p \rangle$ **from** p **at** q
8: **sendto** p : Reply$\langle s_q \rangle$
9: $s_q :=$ Update(s_p, s_q)
10: **end event**

(b) Passive thread

For network size estimation, one random node sets its local state to 1 while all other nodes set their local states to 0. Hence, initially, the global sum is 1 and average is $\frac{1}{n}$, and the variance in local states is large. Executing the aggregation algorithm for a number of cycles decreases the variance of local states but keeps the global sum and average the same. Thus, after *convergence*, a node p can estimate the network size as $\frac{1}{s_p}$. For network size estimation, Update$((s_p, s_q))$ returns $\frac{s_p+s_q}{2}$.

Aggregation [71] achieves up-to-date estimates by periodically restarting the protocol, i.e. local values are re-initialized and aggregation starts again. This is done after a predefined number of cycles γ, called an *epoch*. The states are assumed to have converged at the end of an epoch. Jelasity et. al. also propose mechanisms on how a single node initializes its state to 1 while others initialize their state to 0 in an epoch in a decentralized fashion.

The main disadvantage of Aggregation is that a failure of a single node early in an epoch can severely effect the estimate. For example, if the node with local state 1 crashes after executing a single exchange, 50% of the value will disappear, giving $2 \times n$ as the final size estimate. This issue is further elaborated in Section 4.3.3. Another disadvantage, as we discuss in section 4.3.1, is the requirement of predefining the epoch length γ.

4.2 The Network Size Estimation Algorithm

A naïve approach to estimate the network size in a ring-based overlay would be to pass a token around the ring, starting from, say node i and containing a variable v initialized to 1. Each node increments v and forwards the token to its successor, i.e. the next node on the ring. When the token reaches back at i, v will contain the network size. While this solution seems simple and efficient with respect to number of messages, it suffers from multiple problems. First, it is not fault-tolerant as the node with the token may fail. This will require complicated modifications for regenerating the token with the current value of v. Second, the naïve approach will be quite slow, as it will take $O(n)$ time to complete. Since peer-to-peer systems are highly dynamic, the actual size may have changed by the time the algorithm finishes. Lastly, at the end of the naïve approach, the estimate will be known only to node i which will have to broadcast it to all nodes in the system. Our solution aims at solving all these problems at the expense of a higher message complexity than the naïve approach.

Our goal is to make an algorithm where each node tries to estimate the average inter-node distance, Δ, on the identifier space, i.e. the average distance between two consecutive nodes on the ring. Given a correct value of Δ, the number of nodes in the system can be estimated as $\frac{N}{\Delta}$, N being the size of the identifier space.

Every node p in the system keeps a local estimate of the average inter-node distance in a local variable d_p. Assuming P is the set of all nodes, our goal is to compute $\frac{\sum_{i \in P} d_i}{|P|}$ at each node. The philosophy underlying our algorithm is the observation that at any time, the following invariant should be satisfied: $N = \sum_{i \in P} d_i$.

We achieve our goal by letting each node p initialize its estimate d_p to the distance to its successor on the identifier space. Concretely, we initialize the local state as $d_p = succ(p) \ominus p$, where $\ominus$ represents subtraction modulo N. Note that if the system only contains one node p, then $d_p = N$. Clearly, a correctly initialized network satisfies the afore-mentioned invariant as the sum of the estimates (local states) is equal to N.

We employ a modified gossip-based aggregation algorithm for the exchange of initial local states and convergence of Δ. Since nodes in a structured overlay do not have access to random nodes, we implement the GetRandomNeighbour method in Algorithm 6 by returning a node reached by making a random walk of length h. For instance, to perform an exchange, p sends an exchange request to one of its neighbours, se-

lected randomly from the routing table, with a hop value h. Upon receiving such a request, a node r decrements h and forwards the request to one of its own neighbours, again selected randomly. This process is repeated until h reaches 0, after which the exchange of local states takes place between p and the last node receiving the request.

Given that GetRandomNeighbour returns random nodes, after a number of exchanges (logarithmic number of steps, to the network size, as show in [71]), every node will have $d_p = \frac{\sum_{i \in P} d_i}{|P|}$. Since the variance in the initial local states on all nodes is lower in our case compared to [71], our solution converges in lesser number of exchanges.

In each cycle, on average, each node initiates an exchange once, which takes h hops, and replies to one exchange. Consequently, the number of messages for our modified aggregation algorithm are $h \times n + n$ per cycle. While this is higher than the cost of basic Aggregation ($2 \times n$), our solution does not require a sampling service such as Cyclon [162], which has its own overhead.

4.2.1 Handling dynamism

The protocol described so far does not take into account the dynamic behaviour of large-scale peer-to-peer systems. In this section, we present our solution as an extension of the basic algorithm described in Section 4.2 to handle dynamism in the network. Furthermore, our solution does not require predefining an epoch length for re-initialization.

The basic idea of our solution is that each node keeps different *levels* of estimates, each with a different accuracy. The lowest level estimate is the same as d_n in the basic algorithm. As the value in the lowest level converges, it is moved to the next/higher level. While this helps by having high accuracy in upper levels, it also gives a continuous access to a correct estimated value while the lowest level is re-initialized and converges. Furthermore, the solution uses the lowest level to detect convergence and restart the protocol adaptively, instead of after a predefined interval.

Our solution is given in Algorithm 7 and 8. Each node n keeps track of the current epoch in $epoch_n$ and stores the estimate in a local variable st_n instead of d_n in the basic algorithm. st is a tuple of length l, i.e.

$$st = (st_{l-1}, st_{l-2}, \cdots, st_0)$$

The tuple values are levels corresponding to the accuracy of the estimate at that level. The value at level 0 is the same as d_n in the basic algorithm and has the most recent updated estimate but with high er-

ror, while level '$l-1$' has the most accurate estimate but incorporates updates slowly.

Algorithm 7 Network size estimation (Part I)

```
1:  every δ time units at n
2:      if Converged() and IAmStarter() then
3:          SimpleBroadcast(epoch_n + 1)
4:      else
5:          sendto randNeighbour() : ReqExchange⟨n, epoch_n, st_n, hops⟩
6:      end if
7:  end event

8:  receipt of ReqExchange⟨i, e_i, st_i, h⟩ from m at n
9:      if h > 1 then
10:         h := h − 1
11:         sendto randNeighbour() : ReqExchange⟨i, e_i, st_i, h⟩
12:     else
13:         if epoch_n > e_i then                ▷ initiator i is in an older epoch
14:             sendto i : ResExchange⟨false, epoch_n, st_n⟩
15:         else
16:             trigger ⟨ MoveToNewEpoch | e_i ⟩
17:             sendto i : ResExchange⟨true, epoch_n, st_n⟩
18:             st_n := Update(st_n, st_i)
19:             UpdateSlidingWindow(st_n)
20:         end if
21:     end if
22: end event
```

When a node moves to a new epoch (Algorithm 8, line 13), it promotes the estimates for each level one level up, and then initializes the lowest level (0) using the afore-mentioned technique. The method LeftShiftLevels() moves the estimate of each level one level up, e.g. left shifting a tuple $e = (e_{l-1}, e_{l-2}, \cdots, e_0)$ gives $(e_{l-2}, e_{l-3}, \cdots, e_0, nil)$. Thereafter, the node p initializes its estimate via InitializeEstimate() by setting the level 0 value to $succ(p) \ominus p$.

When two nodes exchange their states, they update their local states by taking average of their own state and the received state. We extend the same anti-entropy principle to tuples. Here, the method Update(a, b) operates on tuples and returns an average of each level, i.e. $(\frac{a_{l-1}+b_{l-1}}{2}, \frac{a_{l-2}+b_{l-2}}{2}, \cdots, \frac{a_0+b_0}{2})$.

To incorporate changes in the network size due to churn, we restart the algorithm by moving to a new epoch. Instead of restarting after

Algorithm 8 Network size estimation (Part II)

```
1: receipt of ResExchange⟨updated, e_m, st_m⟩ from m at n
2:     if updated = false then
3:         trigger ⟨ MoveToNewEpoch | e_m ⟩
4:     else
5:         if epoch_n = e_m then
6:             st_n := UPDATE(st_n, st_m)
7:         end if
8:     end if
9: end event

10: receipt of DeliverSimpleBroadcast⟨e_m⟩ from m at n
11:     trigger ⟨ MoveToNewEpoch | e_m ⟩
12: end event

13: upon event ⟨ MoveToNewEpoch | e ⟩ at n
14:     if epoch_n < e then          ▷ join new epoch if in an older epoch
15:         LEFTSHIFTLEVELS()
16:         INITIALIZEESTIMATE()
17:         epoch_n := e
18:     end if
19: end event
```

a predefined number of cycles (as in [71]), we do so adaptively by analyzing the variance of a history of estimates. We let the lowest level *converge* such that the change in estimate is negligible for consecutive cycles, and then restart. This duration may be larger than a predefined epoch length γ or less, depending on the system-wide variance of the value being estimated, and the actual network size. We achieve adaptivity by using a sliding window at each node. Each node stores multiple values of the lowest level estimate for consecutive cycles in a sliding window, replacing estimates of older cycles by newer cycles. If the coefficient of variance of the sliding window is less than a desired accuracy, e.g. 10^{-2}, the value is considered converged. This logic is embedded in the method CONVERGED() in Algorithm 7.

Once the estimate at the lowest level is considered to have converged based on the sliding window, a new epoch can be started. There are different mechanisms of deciding which node should restart the protocol, denoted by the method IAMSTARTER(). One way is, as used in [71], that each node restarts the protocol with probability $1/\hat{n}$, where $\hat{n}$ is the current estimated network size. Given a reasonable estimate in the

previous epoch, this will lead to one node restarting the protocol with high probability. Multiple nodes starting an epoch does not effect the correctness of the estimate, and only results in redundant messages. [1] An alternate mechanism is that the node p which has $0 \in [p, p.succ)$ restarts the protocol by starting a new epoch. In our evaluation, we use the first method.

Once a new epoch starts, all nodes should join it quickly. Aggregation [71] achieves this by the logarithmic epidemic spreading property of random networks. Since we do not have access to random nodes in structured overlays, we use a simple broadcast scheme [47] for this purpose, which is both inexpensive ($O(n)$ messages) and fast ($O(\log n)$ steps). The broadcast is best-effort, as even if it fails, the new epoch number is spread through exchanges.

When a new node joins the system, it starts participating in the size estimation protocol when the next epoch starts. This happens either when it receives the broadcast, or its predecessor initializes its estimate. Until then, it keeps forwarding any requests for an exchange to a randomly selected neighbour.

Handling churn in our protocol is simpler and less expensive, in terms of bandwidth consumption, than Aggregation [71]. Instead of running multiple epochs concurrently (as in [71]), we rely on the fact that a crash in our system does not effect the end estimate as much as in [71]. The reason is as follows. In our protocol, since node identifiers are uniformly distributed in structured overlays, the variance of the initial local states – distance between a node and its successor – is low. Thus, a failure of a node in the beginning of an epoch will not have severe affect on the final estimate. Furthermore, since the variance is low to begin with after initialization of local states, our algorithm converges in fewer number of cycles. The effect of failures on our algorithm is explored in detail in Section 4.3.3.

4.3 Evaluation

To evaluate our solution, we implemented the Chord [152] overlay in an event-based simulator [144]. We simulate two network sizes, 5000 and 10^5. The simulations for network size of 5000 use the King dataset [59] for message latencies between nodes. Since the King dataset is not available for a 5000 node topology, we derive the 5000 node pair-wise la-

[1] On the contrary, multiple nodes starting an epoch in [71] is problematic since only one node should set its local estimate to 1 in an epoch. Consequently, an epoch has to be marked with a unique identifier.

tencies from the distance between two random points in a Euclidean space [92]. The mean round trip time remains the same as in the King dataset. For simulating a larger network size, we start 10^5 nodes and use random message latencies between nodes with exponential distribution and mean of 5 simulation time units. In the figures, $\delta = 8 * mean\text{-}com$ means the cycle length is 8×5.

The main idea behind our size estimation algorithm is for each node to estimate the average inter-node distance d on the ring identifier space. In the first set of experiments, we measure the *error* in the estimate of d on all nodes versus the actual inter node distance ($\frac{N}{n}$). Concretely,

$$error = \frac{1}{n} \sum_{i=1}^{n} |d_i - \frac{N}{n}|$$

We initialize the local states on all nodes and measure the error over cycles. We evaluate the number of cycles needed for convergence, i.e. the error to be small, when using various values of δ, and number of hops.

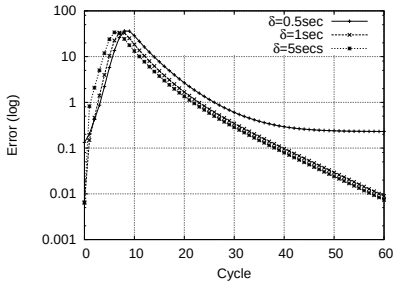
In the second set of experiments, we compare the estimate of the network size by our algorithm to the actual network size under different churn scenarios. We also compare the robustness of our solution to Aggregation.

4.3.1 Epoch length

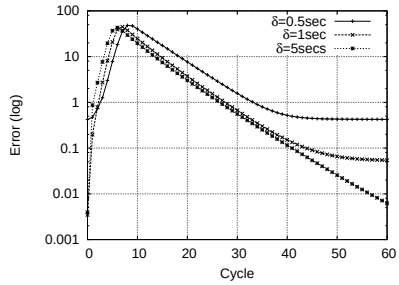
In this section, we evaluate the effect of the time period for initiating an exchange, δ, on the convergence of the algorithm and hence, the epoch length. We measure the error periodically after every cycle. Figure 4.1 shows the results when using King latencies and a network size of 5000, and Figure 4.2 shows results for exponentially distributed latencies with mean 5 and a network size of 10^5.

Our results show that when the ratio between the communication delay and δ is high, e.g. $\delta = 0.5\ secs$ (Fig. 4.1) and $8 * mean\text{-}com$ (Fig. 4.2), the aggregate converges slowly and to a value with higher error. For cases where the ratio is low, e.g. $\delta = 5\ secs$ (Fig. 4.1) and $24 * mean\text{-}com$ (Fig. 4.2), the convergence rate is faster and the converged value has lower error. The reason is that when δ is short, the expected number of exchanges per cycle do not occur. This problem is further aggravated in our solution since we use a random walk to find a random node in the network.

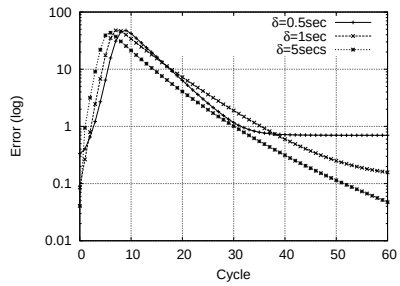
Since δ effects convergence rate and accuracy, using a predefined number of cycles as the epoch length (as in [71]) can lead to an estimate



(a) hops = 0



(b) hops = 3



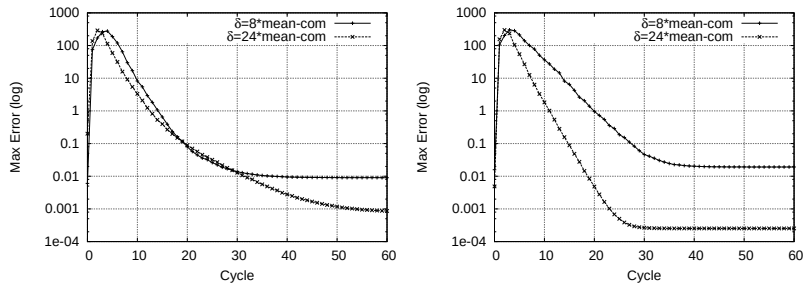
(c) hops = 6

Figure 4.1: Error for the estimate of inter-node distance d in the system for a network size of 5000 nodes.

with large error. Hence, our solution uses adaptive epoch lengths. Another benefit of using an adaptive approach as ours is that the protocol may converge much before a predefined epoch length, thus sending messages in vain between the time the algorithm converged and the end of the epoch. If the protocol is restarted as soon as the estimate has converged (as in our solution), these extra cycles are used to get updated aggregate value, hence reflecting churn effects in the estimate faster.

4.3.2 Effect of the number of hops

In this section, we discuss the effect of the number of hops taken to find a random node to exchange states. Figure 4.3 shows convergence of the algorithm for different values of number of hops, h. Our results show that when $h > 0$, the error converges to a smaller value, and the convergence rate is faster compared to $h = 0$. The reason being that when



(a) hops = 0 (b) hops = 4

Figure 4.2: Error for the estimate of inter-node distance d in the system for a network size of 10^5 nodes.

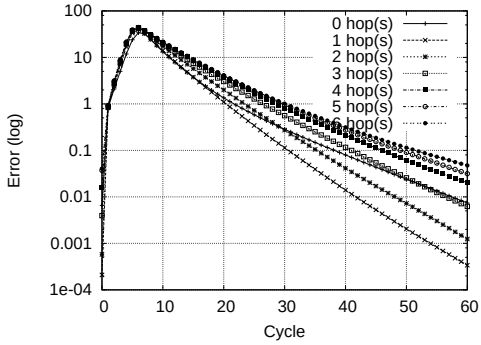
$h = 0$, nodes will exchange their local states only with their neighbours, which slows down the diffusion process. On the other hand, using $h > 0$ gives better access to random nodes in the overlay to spread local states. Furthermore, using very large values of h might not be helpful either as it can prevent the expected number of exchanges to take place in each cycle.

4.3.3 Churn

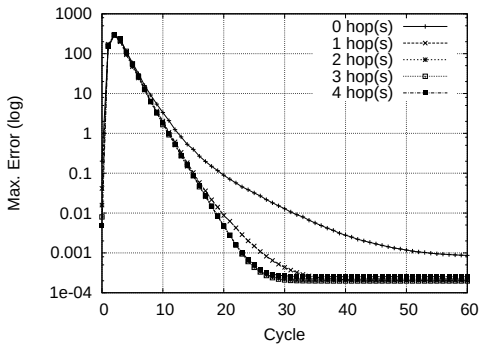
Flash crowds

The basic idea behind aggregation is to reduce the variance in local states of all nodes over cycles by exchanging values, while keeping the total sum of local states the same as when local states were initialized in the beginning of the epoch. The main reason why the total sum may change at the end of the epoch from the beginning is node failures. When a node fails, its local state is lost, reducing the total sum. This affects the converged local states and hence, the estimate of the network size. On the other hand, a node join can be prevented from influencing the aggregation process by preventing it from joining an in-progress epoch.

The variance in local states in the beginning of an epoch, i.e. in the first few cycles after local states are initialized, is high in [71] compared to our solution. Thus, a failure of a node early in an epoch can result in a worse estimate when using [71] than our solution. We simulate and compare such a scenario in this section.



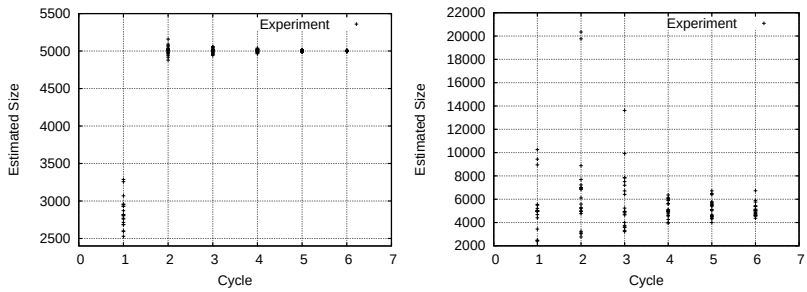
(a) n = 5000, $\delta = 5$ seconds



(b) n = 10^5, $\delta = 24 * mean\text{-}com$

Figure 4.3: Comparison of the error in the estimate of inter-node distance d for various values of the number of hops.

To mimic extreme situations, e.g. a network partition, we evaluate a scenario where a massive number of nodes fail simultaneously. In each experiment, we fail 50% of the nodes at a given cycle of an epoch and measure the network size estimated at the end of the epoch. Our results, when the simultaneous failure occurs at various cycles, is shown in Figure 4.4. The x-axis shows the cycle at which the massive failure occurred, and the y-axis shows the converged estimate. Our modified aggregation solution (Figure 4.4(a)) is not as severely affected by the sudden massive failure as the original Aggregation algorithm (Figure 4.4(b)). In fact, in some of the Aggregation experiments, the estimate converged to infinity (not shown in the figure). This happens when all the nodes with non-zero local estimates fail. For our solution,



(a) Our modified aggregation (b) Aggregation by Jelasity et. al. [71]

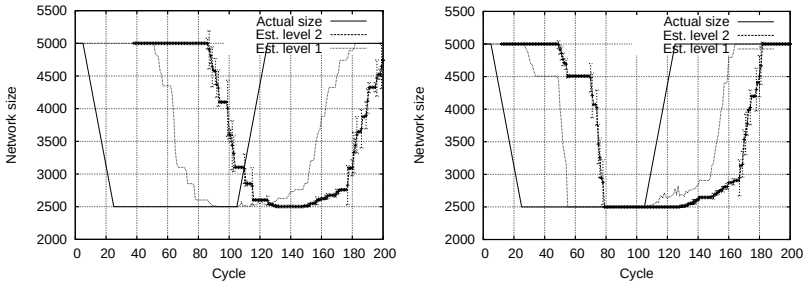
Figure 4.4: Estimated size when 50% of the nodes out of 5000 fail suddenly. X-axis gives the cycle at which the sudden failure occurred in the epoch.

the effect of a sudden simultaneous failure is already negligible if the nodes crash after the third cycle.

Continuous churn

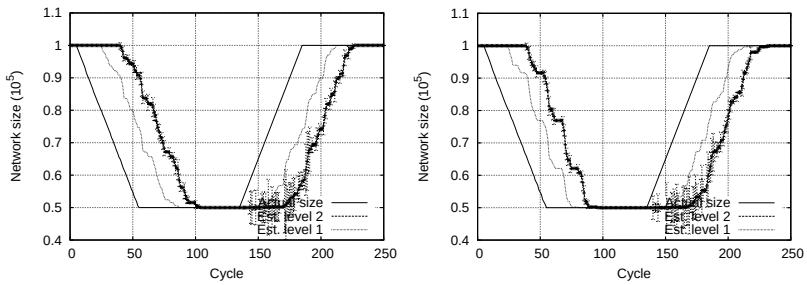
In this section, we evaluate churn scenarios where nodes join and fail throughout the duration of the experiment. We simulate churn cases where the total network size gradually decreases and then increases, and measure the estimated network size. We keep track of the estimate at 2 levels (see Section 4.2). Figure 4.5 and 4.6 show the results when 50% nodes fail within a few cycles and afterwards, 50% nodes join within a few cycles. The graphs show how the estimation follows the actual network size, with some delay before the actual network size is reflected in the estimate. The standard deviation of the estimated network size at level 2 is shown as vertical bars, which depicts that all nodes estimate roughly the same size. The standard deviation is high only when a new epoch starts, because while evaluating the mean and standard deviation, some nodes have moved to the new epoch, while others are still in the previous epoch. The estimate at level 1 converges to the actual size faster than level 2, but the estimate has higher variance as the standard deviation for level 1 (omitted) is higher than for level 2.

Next, we evaluate the affect of the hop count h on how fast the estimate follows the actual network size. Figure 4.7 shows the results when the network size is halved in a few cycles. Compared to $h = 0$, higher values of h follow the trend of the actual size faster.



(a) hops = 0 (b) hops = 1

Figure 4.5: Mean estimated size by each node with standard deviation using King dataset latencies.



(a) hops = 0 (b) hops = 1

Figure 4.6: Mean estimated size by each node with standard deviation using exponentially distributed latencies with mean 5 time units.

Finally, we simulate a network of size 4500 and evaluate our algorithm under continuous churn, with a mix of joins and failures. In each cycle, we fail some random nodes and join new nodes. Figure 4.8 shows our results. The plotted dots correspond to the converged mean estimate after 15 cycles for each experiment. The x-axis gives the percentage of churn events, including both failures and joins, that occur in each cycle. For instance, 10% means that $4500 \times \frac{10}{100} \times 15$ churn events occurred before the plotted converged estimated network size. Our results show that the algorithm handles continuous churn reasonably well as the estimate is close ($5 - 10\%$) to the actual network size..

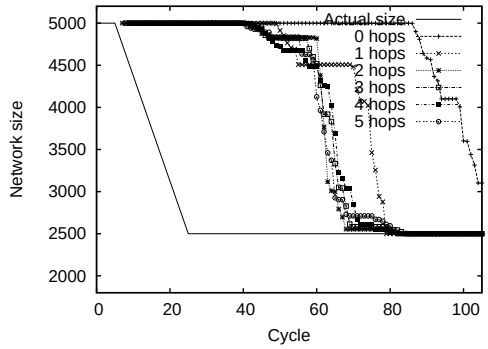


Figure 4.7: Mean estimated size for different values of h for level 2. The figure compares how fast the size estimation follows the real network size for various h.

4.4 Related Work

Network size estimation in the context of peer-to-peer systems is challenging as these systems are completely decentralized. Furthermore, nodes may fail anytime, the network size may vary drastically over time, and the estimation algorithm should continuously update the estimation to reflect the current number of nodes.

Merrer et. al. [107] have compared three existing size estimation algorithms, Sample & Collide [105], Hops Sampling [78] and Aggregation [71], which are representative of three main classes of network size

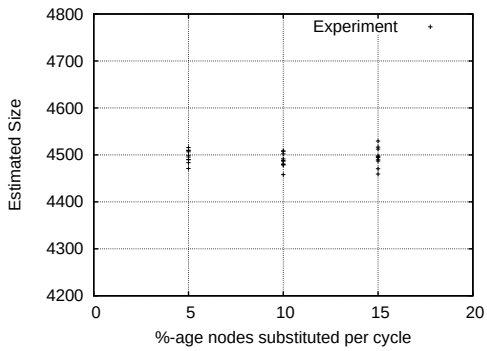


Figure 4.8: Estimated network size for 4500 nodes under continuous churn. X-axis gives the percentage of churn events (joins+failures) that occur in each cycle.

estimation algorithms. Their study yields that although Aggregation is expensive, it produces the most accurate results. Aggregation also has the additional benefit that the estimate is available on all nodes compared to only at the initiator in the case of Sample & Collide and Hops Sampling. Our work can be seen as an extension of Aggregation, to handle its shortcomings and extend it to non-random topologies, such as structured overlay networks.

The work by Horowitz et. al. [64] is similar to ours in the sense that they also utilize the structure of the overlay for the estimation. They use a localized probabilistic technique to estimate the network size by maintaining a structure: a logical ring. Each node estimates the network size locally based on the estimates of its neighbours on the ring. While their technique has less overhead, the estimates are not accurate, the expected accuracy being in the range $\frac{n}{2} \cdots n$. Their work has been extended by Andreas et. al. [17] specifically for Chord, yet the extended work also suffers similar inaccuracy range for the estimated size. Mahajan et. al. [102] also estimate the network size through the density of node identifiers in Pastry's leafset, yet they neither prove any accuracy range, nor provide any simulation results to show the effectiveness of their technique.

Kempe et. al. [77] have suggested a gossip-based aggregation scheme, yet their solution focuses only on push-based gossiping. Using push-based gossiping complicates the update and exchange process as a normalization factor needs to be kept track of. On the same, as noted by Jelasity et. al. [71], push-based gossiping is problematic when the underlying directed graph used is not strongly connected. Thus, we build our work on push-pull gossip-based aggregation. Similarly, to estimate the network size, Kempe et. al. also propose that one node should initialize its weight to 1, while the other nodes initialize to weight 0, making it highly sensitive to failures early in the algorithm.

The authors of Viceroy [103] and Mercury [15] mention that a nodes distance to its successor can be used to calculate the number of nodes in the system, but provide no reasoning that the value always converges exactly to the correct value, and thus that their estimate is unbiased.

CHAPTER 5

Lookup Inconsistencies

Structured overlays networks provide a lookup service (see Section 2.1.2) for Internet-scale applications, where a lookup maps a key to a node in the system. The node mapped by the lookup can then be used for data storage or processing. Distributed Hash Tables (DHTs) [32, 40] use an overlay's lookup service to store data and provide a put/get interface for distributed systems. Since the lookups are "best-effort", DHTs built on overlays typically guarantee eventual consistency [81, 38]. In contrast, many distributed systems, such as distributed file systems [131] and distributed databases [166], require stronger consistency guarantees. These systems generally rely on services such as consensus [87] and atomic commit [100].

In this chapter, we discuss how anomalies in the routing level, causing *lookup inconsistencies*, result in inconsistencies at the data level. We study the causes, and frequency of occurrence, of lookup inconsistencies under different scenarios in a DHT. We discuss techniques that can be used to decrease the effect of lookup inconsistencies at the cost of reduced availability.

Introduction Ring-based overlays use consistent hashing [74] to map keys to nodes under churn. Existing nodes take over key responsibilities of inaccessible/failed nodes, and newly joined nodes take over a fraction of the responsibilities of existing nodes (see Section 2 for details). While the overlay updates its routing pointers to cater for churn, routing anomalies can arise temporarily. Apart from churn, incorrect

routing pointers can also arise due to incorrect behaviour of *failure detectors*.

Failure detectors A failure detector is a module used by a node to determine if another node is alive or has failed. Failure detectors are defined by two properties: *completeness* and *accuracy* [23]. In a crash-stop process model, completeness requires the failure detector to eventually detect all crashed/failed nodes. Accuracy relates to the mistake a failure detector can make to decide if a node has crashed or not. A perfect failure detector is accurate all the times, *i.e.*, it never detects an alive node as failed. Failure detectors are generally implemented using heartbeats, where a node periodically sends heartbeat messages to another node. If a node p does not receive a heartbeat from another node q within a certain time period, the failure detector at p detects q as failed.

Structured overlays are aimed to operate on *asynchronous* networks. Informally, a network is asynchronous if there is no bound on the time taken by a message to be delivered to the destination node. Thus, no timing assumptions can be made in such networks. Due to the absence of timing restrictions in an asynchronous model, it is impossible for a failure detector to determine if another node has actually crashed or is very slow to respond to heartbeats. This may violate the accuracy property of a failure detector, giving rise to inaccurate detection of node failures.

For our work, we assume an imperfect failure detector that is complete but only probabilistically accurate. A node p sends a ping message to its neighbours at regular intervals. If p receives an acknowledgment of the ping message from a neighbour before a timeout, the neighbor is considered to be alive. Not receiving an acknowledgment within the timeout implies the neighbor has crashed. The timeout is chosen to be much higher than the typical round-trip time between the two nodes. This model of failure detector is similar to the baseline algorithm used by Zhuang et. al [165]

5.1 Consistency Violation

Temporary anomalies on the routing level can lead to lookup inconsistencies. In this section, we define a consistent configuration of an overlay, and discuss how violating a consistent configuration can give rise to lookup inconsistencies. In turn, we show how lookup inconsistencies can lead to inconsistencies on the data level.

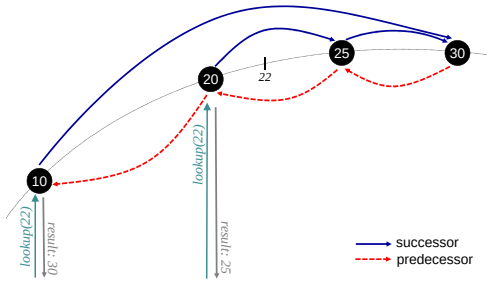


Figure 5.1: An inconsistent configuration. Due to imperfect failure detection, 10 suspects 20 and 25, thus pointing to 30 as its successor.

A *configuration* of an overlay is a set of all nodes and their pointers to neighboring nodes, including successor, predecessor and fingers. An overlay, and thus its configuration, evolves by either changing a pointer, or adding/removing a node. For our purposes, we define a consistent configuration of the overlay as follows.

Definition 1. *A configuration of the system is consistent if, in that configuration, lookups made for the same key from different nodes, return the same node.*

In a configuration where consistency is violated, we can have *inconsistent lookups i.e.* multiple lookups for the same key may be resolved to different nodes in that configuration. In other words, multiple nodes are deemed responsible for the same key. Lookup consistency may be violated if some node's successor pointers do not reflect the current ring structure[1].

Figure 5.1 illustrates an inconsistent configuration where key responsibilities are not well-defined, and hence, multiple lookups for key 22 can return different results. The figure shows a part of the ring identifier space with nodes and their successor and predecessor pointers. Such a configuration may arise when, due to inaccuracy of the failure detector, 10 falsely suspects 20 and 25 to have failed. Thus, node 10 believes that the next (clockwise) alive node on the ring is 30, and points to it as its successor. Subsequently, a lookup for key 22 arriving at 10 will return 30 as the responsible node for key 22, whereas a lookup arriving at 20 will return 25.

[1]For our purposes, we define a consistent configuration based on lookup results. A possible alternate definition can be based on correct successor and predecessor pointers. We do not use such a definition as it may or may not lead to inconsistent lookups and inconsistencies on the data level

Lookup inconsistencies can give rise to data inconsistencies. For instance, assume the value stored under key 22 is v in the inconsistent configuration shown in Figure 5.1. Next, an update operation is initiated to store value v' against key 22. Such an update uses a lookup to find the node responsible for key 22. The lookup can result in either node 25 or 30, hence the updated value v' will be stored at either node 25 or 30. Assume the update is stored at node 25. Now, a read operation for data with key 22 can potentially return inconsistent/old value v, instead of value v', if the lookup reaches the node that didn't receive the update, *i.e.* node 30.

Network partitions can cause imperfect failure detections, where nodes in one partition falsely suspect the nodes in the other partition as failed. As false failure detections can lead to overlapping responsibilities, network partitions lead to overlapping responsibilities as well. An example of such a scenario is when a network partition leads the overlay to the divided into two overlays. Here, nodes from one partition will falsely detect nodes in the other partition as failed. Each partitioned overlay will define responsibilities for the whole identifier space, hence leading to an overlap of the entire set of keys.

Apart from inaccurate failure detectors, an inconsistent configuration can also arise due to high churn rates. Some such scenarios are discussed in [47].

5.2 Inconsistency Reduction

In this section, we present a notion, called *local responsibility*, for reducing lookup inconsistencies. Furthermore, we discuss how the effect of lookup inconsistencies is reduced in *quorum-based* algorithms.

5.2.1 Local Responsibility

In this section, we define local and global responsibility for keys, and discuss how using the notion of local responsibility can reduce inconsistencies.

Definition 2. *A node n is said to be "locally responsible" for a key k if the key k is in the range between its predecessor and itself, i.e.* $k \in (n.pred, n]$.

Definition 3. *A node n is said to be "globally responsible" for a key k if n is the only node in the overlay configuration that is locally responsible for key k.*

It follows from Definition 2 that the local responsibility of a node (and indirectly keys) changes whenever its predecessor pointer is changed.

This is different from the notion of responsibility based on lookups (as described in Section 2.1.2 and used in Section 5.1), which uses successor pointers only. We modify the lookup operation of an overlay such that a lookup for key k always terminates at and returns the locally responsible node n for k, i.e. $k \in (n.pred, n]$[2]. Thus, before returning the result of a lookup, the node checks if it is locally responsible for the key being looked up. In case the node is not locally responsible, it can either forward the request clockwise or anti-clockwise, or ask the initiator of the lookup to retry.

Using our modified lookup operation that uses local responsibilities in Definition 1, it follows that a configuration is consistent if there is a globally responsible node for each key [3]. Similarly, the local responsibility for a key k is consistent if there is a node globally responsible for k. This implies that if a node is globally responsible for k, then lookup inconsistencies cannot occur with our modified lookup operation. The reason being that a lookup for k will either fail (due to node failures), or end up at the node that is globally responsible for k. Since there is only one such node, multiple lookups will end up at the same node, and thus, there will be no lookup inconsistencies.

Although the configuration depicted in Figure 5.1 is inconsistent according to Definition 1, yet it is consistent with respect to lookups with local responsibilities. The reason being that the local responsibilities do not overlay in the configuration. Here, when a lookup for key 22 arrives at 20, instead of replying, it will forward the lookup to node 30 due to our modified lookup operation. Since 30 is not locally responsible for 22, it will either notify the lookup initiator to retry or forward the lookup to its predecessor 25. Since node 25 is locally responsible for key 22, it will reply to the lookup as resolved to 25. If a lookup arrives at node 20, it will also forward the lookup to node 25. Thus, multiple lookups will always return the same result. This will lead data operations (read/write) to operate on the same node, and hence, achieve data consistency.

If a node has an incorrect predecessor pointer, the range of keys it is locally responsible for can overlap with another node's key range. In such a case, there will be multiple nodes locally responsible for the same key leading to inconsistency. An example of such a configuration is shown in Figure 5.2. Here, both node 25 and 30 are locally responsible for 22. Such a configuration may arise if node 10 falsely suspects node 20 to have failed, while both nodes 20 and 30 falsely suspect node 25.

[2] instead of terminating at a node m such that $k \in (m, m.succ]$, and returning $m.succ$

[3] Ensuring that there is only one globally responsible node for each key is non-trivial, and is the focus of Chapter 6

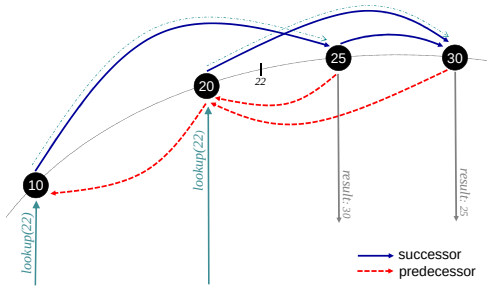


Figure 5.2: An inconsistent configuration with respect to *local responsibilities*. Due to imperfect failure detection, node 10 suspects node 20 while node 20 suspects node 25.

A lookup for key 22 that arrives at node 10 will be forwarded to node 25, which will return the result of the lookup as 25 since it is locally responsible for key 22. On the other hand, a lookup for key 22 that arrives at node 25 will be forwarded to 30, which is also locally responsible for key 22. Hence, even when using local responsibilities, configurations can be inconsistent as multiple lookups for the same key can resolve to different nodes. As we showed before, inconsistent configurations can lead to data inconsistency as well.

Figure 5.2 shows that the method of local responsibilities does not completely solve the problem of inconsistencies, but it decreases inconsistencies. This is mainly because without local responsibility, only one node doing inaccurate failure detections is enough to introduce inconsistencies. This can easily happen on the Internet as a node may have a transient link failure, slow connection to some other nodes, or be overladed. On the other hand, when using local responsibilities, multiple nodes have to do simultaneous inaccurate failure detections to result in routing and data inconsistencies. We measure and discuss decrease in lookup inconsistencies when using local responsibilities in Section 5.3.

Key Availability

Brewer's conjecture [19, 51] states that it is impossible for a web service to provide all the three properties of consistency, availability and partition-tolerance. The same principle applies to our technique of local responsibility as well. While using local responsibilities can reduce inconsistencies, it can potentially results in keys becoming unavailable. Section 5.3 elaborates, via simulation, that using the method of local responsibilities increases consistency at the cost of availability. In this

section, we first define availability of a key and then show how a key becomes unavailable when using local responsibilities.

Definition 4. *In a configuration, a key k is available if there exists a reachable node n such that n is locally responsible for k.*

Here, a node n is *reachable* in a configuration if there exists a node m such that n is the successor of m, *i.e.* $m.succ = n$ and $n \neq m$[4]. The notion of a node n being reachable captures the fact that a lookup can resolve to n.

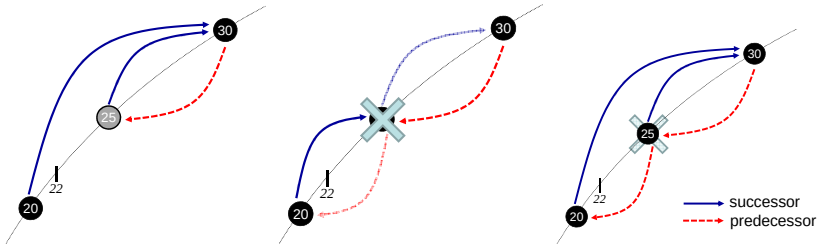
Availability of a key in an overlay is affected by both churn and inaccurate failure detectors. When a node joins the system, it changes the responsibility key range of its successor. This leads to temporary unavailability of some keys. Figure 5.3(a) shows a configuration where node 25 joins the overlay. Node 30 points to 25 as its predecessor, thus making key 22 unavailable as no node is locally responsible for 22. Key 22 remains unavailable until 20 runs periodic stabilization and sets $20.succ = 25$, and node 25 sets $25.pred = 20$. In Chapter 6, we reduce such key unavailability by keeping the responsibility of keys with a node until another node explicitly asks for a handover of the keys.

Failure of a node leads to temporary unavailability of keys until the failure is detected, and pointers are updated. Such a case is shown in Figure 5.3(b) where node 25 crashes. Key 22 remains unavailable until node 20 detects the failure of 25, sets $20.succ = 30$, and runs periodic stabilization with node 30 resulting in setting $30.pred = 20$.

Inaccuracy of failure detectors may also lead to unavailability of keys. This occurs when a node falsely suspects its successor and removes its pointer to the suspected node. Keys for which the suspected node is locally responsible will temporarily become unavailable as it becomes unreachable. Such a scenario is shown in Figure 5.3(c) where node 20 suspects 25 leading to unavailability of key 22 as node 25 becomes unreachable.

Systems that implement atomic join and graceful leaves such as DKS [7] and CATS (Chapter 6) will alleviate unavailability in the join cases (e.g. Figure 5.3(a)), but not in failure cases (e.g. Figure 5.3(b) and 5.3(c)). To handle failures, the data has to be replicated on multiple machines, which leads to the requirement of keeping the replicas consistent. We address such cases in more detail in Chapter 6 and Chapter 7.

[4] or m has a finger pointing to n. We ignore reachability due to a finger pointer as lookups are resolved through successor pointers



(a) 25 joins (b) 25 fails (c) 20 falsely suspects 25

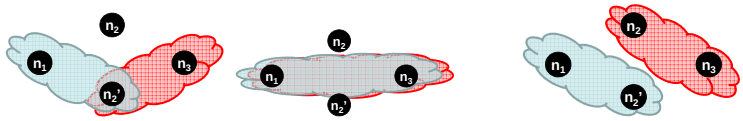
Figure 5.3: Unavailability of a key with respect to local responsibilities for various scenarios.

5.2.2 Quorum-based Algorithms

In this section, we show that using quorum-based algorithms can reduce the effect of lookup inconsistencies, thus increasing the probability of data consistency. While such a usage increases the chances of data consistency, the probability of data inconsistency is still non-zero. To achieve absolute data consistency, we need additional mechanisms for maintaining group membership. This is the topic of Chapter 6.

Like any distributed systems, overlays/DHTs replicate data on different nodes to increase availability and prevent loss of data. In this chapter, we assume key-based replication, e.g. symmetric replication [48], where an item is replicated by storing it against various keys (see Chapter 2 for details). The replicas of a data item can be found by making lookups for the keys of the data item.

Current implementations of overlays and replication schemes in overlays rarely provide linearizability [63] on their replicas, e.g. Dynamo [38], Cassandra [81]. As overlays are dynamic systems, data maintenance mechanisms have to cope with failures and temporary unavailability of nodes. Quorum-based algorithms seem to be well suited as they can cope with unavailability of a number of replicas as long as a quorum of replicas is available. Owing to these properties, quorum-based algorithms are widely used in overlays (e.g. in Dynamo, and Cassandra), and are considered in this section. Without loss of generality, we focus on majority-based quorum algorithms.



(a) Overlapping majority sets. (b) Disjoint majority sets.

Figure 5.4: Using key-based replication, a data item is stored under 3 different keys for a replication degree of 3. Assuming n_1 and n_3 are responsible for two such keys, while due to an inconsistent configuration, both n_2 and n_2' are responsible for the third key. The majority sets may then overlap or not.

Key-based Consistency with Majority-based Algorithms

In a DHT, data is accessed by making a lookup for the key of the data. As shown before, the responsibility for a certain key might be inconsistent in the system. Without using replication (replication degree=1), if there is a lookup inconsistency, there will be inconsistency in data. For instance, in a configuration shown in Figure 5.2, an update for item to be stored under key 22 might end on 25, while a subsequent read for the same key may access node 30 due to lookup inconsistency. Such a read will return old data, thus leading to data inconsistency and violating linearizability [63]. On the other hand, in key-based replication, even with lookup inconsistencies, there is a high probability that consistency of data is maintained.

Consider a system with replication degree three, where a data item to be stored with key k is stored under keys $\{k_1, k_2, k_3\}$ [5]. Nodes n_1 and n_3 are responsible for k_1 and k_3, while due to lookup inconsistency, two nodes n_2, n_2' are responsible for k_2 [6]. Any operation (read or write/update) for k has to operate on a majority *i.e.* two nodes. Consistency in the afore-mentioned case depends on the way majorities are chosen. Figure 5.4(a) shows cases where majorities for multiple updates overlap, thus data remains consistent. On the other hand, Figure 5.4(b) shows a case where the majorities do not overlap, hence operations will happen on different non-overlapping majority sets, thus leading to data inconsistency.

Using majority-based quorum algorithms increases the chances of data consistency in DHTs since even with lookup inconsistencies, multiple overlapping majorities exist that will lead to data consistency.

[5] in general, $k = k_1$

[6] just like nodes 25 and 30 are responsible for 22 in Figure 5.2

Modeling the Probability for Disjoint Majority Sets As discussed in Section 2, the correctness condition for quorum-based algorithms is that all operations overlap on at least one node. In this section, we analytically derive the probability that two operations work on disjoint/non-overlapping majority sets given the system configuration is the same for the two operations. Such a probability directly translates into violation of the quorum requirement, thus leading to data inconsistency.

We model the probability for disjoint majority sets by using the counting principle. The probability of disjoint majority sets is the ratio between the number of possible disjoint majority sets and the number of all combinations of majority sets that two operations in one configuration can include. For the sake of simplicity, we assume that for a responsibility inconsistency in the configuration, only two nodes are responsible for the inconsistency, i.e., for an inconsistency of a key, two nodes are responsible for the key.

Consider an overlay with replication degree r (where $r > 0$), size of the smallest majority set denoted by m (where $m = \lfloor \frac{r}{2} \rfloor + 1$), and a configuration with i number of responsibility inconsistencies (where $i > 0$). Equation (5.1) gives the number of all possible combinations for two majority sets, denoted as $T_{i,r}$. Here, j is the number of inconsistencies included in a majority set. Since each inconsistency creates two possibilities to select a node, we multiple with 2^j.

Equation (5.2) gives, $A_{i,r}$, the number of possible combinations for two disjoint majority sets m_1 and m_2. We compute $A_{i,r}$ by choosing a majority set m_1 and calculating every possible majority set m_2 that is disjoint from m_1. Here, j denotes the number of inconsistencies that are included in m_1. m_2 can share a subset of these j inconsistencies, and additionally include up to $i - j$ remaining inconsistencies. The derived formula is similar to a hyper-geometric distribution.

If p is the probability of an inconsistent responsibility, and assuming inconsistencies are independent, pi_r in Equation (5.3) gives the probability that two subsequent operations in one configuration work on disjoint majority sets.

Figure 5.5 plots the probability of having disjoint majority sets, $\mathcal{P}$, for two operations calculated by $\frac{A_{i,r}}{T_{i,r}}$. It shows how $\mathcal{P}$ depends on the system's replication factor r and on the number of inconsistencies i in the replica set. An important observation is that an even replication degree reduces $\mathcal{P}$ considerably. The reason for such a behaviour is that for majority-based quorums with even replication degree, any two quorums overlap over at least two replicas (say r_1 and r_2). Due to lookup inconsistencies, even if quorums do not overlap at r_1, there is a signifi-

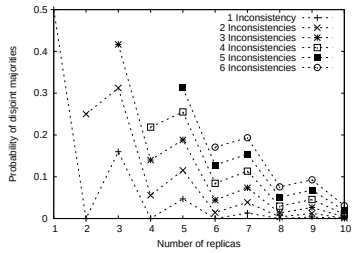


Figure 5.5: Probability of getting disjoint majority sets (y-axis) for a replica set given replica degree (x-axis) and the number of lookup inconsistencies for the keys in the replica set.

cant chance that they will overlap at r_2. This reduces the probability of getting disjoint majority sets.

$$T_{i,r} = \left(\sum_{j=max(m-r+i,0)}^{min(i,m)} \binom{r-i}{m-j} \binom{i}{j} 2^j \right)^2 \tag{5.1}$$

$$A_{i,r} = \sum_{j=\max(m-r+i,0)}^{\min(m,i)} \sum_{k=\max(2m+i-r-2j,0)}^{\min(m,i-j)} 2^{k+j} \times \binom{r-i}{m-j} \binom{i}{j} \times \binom{r-m-i+2j}{m-k} \binom{i-j}{k} \tag{5.2}$$

$$pi_r = \sum_{i=1}^{r} (1-p)^{r-i} p^i \frac{A_{i,r}}{T_{i,r}} \tag{5.3}$$

As lookup consistency cannot be guaranteed in a overlay, even with using local responsibilities and quorum techniques, it is impossible to ensure data consistency. However the violation of lookup consistency when using the afore-mentioned techniques is a result of a combination of very infrequent events. In the following section we present simulation results that measure the probability of lookup inconsistencies and the affects of using local responsibilities and quorums on lookup inconsistencies.

5.3 Evaluation

In this section, we evaluate the frequency of occurrence of lookup inconsistencies, overlapping responsibilities and unavailability of keys resulting from unreliable failure detectors and churn. The metrics of interest are the fraction of nodes that are correct, i.e. do not contribute to inconsistencies, and the percentage of keys available.

For our simulations, the accuracy of a failure detector is defined by its probability of working correctly. The probability of a *false positive* (detect an alive node as dead) is the probability of inaccuracy of failure detectors. A failure detector with probability of false-positive equal to zero is a perfect failure detector. In our experiments, we implemented failure detectors in two styles: *independent* and *mutually-dependent*. For independent failure detectors, two separate nodes falsely suspect the same node as dead independently. Thus, if a node p is a neighbor of both q and r, the probability of q detecting p as dead is independent of the probability of r detecting p as dead. For mutually-dependent failure detectors, if a node p is suspected dead, all nodes doing detection on p will detect p as dead with higher probability, representing a positive correlation between suspicions of different failure detectors. This may be similar to a realistic scenario where due to p, or the network link to p, being slow, nodes do not receive ping replies from p thus detecting it as dead. In the afore-mentioned case, if p is suspected, both q and r will detect it as failed with higher probability than the probability of false-positive. Unless specified, we use independent failure detectors.

Our simulation scenario had the following structure: Initially, we successively joined nodes into the system until we had a network with 1024 nodes. We then started to gather statistics by regularly taking snapshots of the system after every 100 time units. A snapshot is equivalent to freezing the system state. In each snapshot, we counted the number of nodes contributing to lookup inconsistency and overlapping responsibilities. For the experiments with churn, we introduced node joins and failures between taking the snapshots. We varied the accuracy of the failure detectors from 95% to 100%, where 100% means a perfect failure detector. This range seems reasonable, since failure detectors on the Internet are usually accurate 98% of the time [165]. The results presented in the graphs are averages of 1800 snapshots and 30 different seeds.

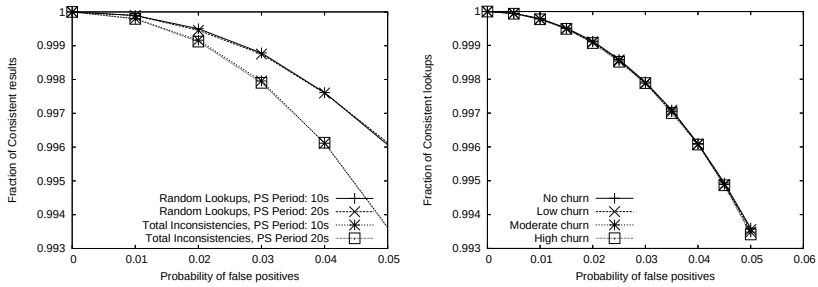
Lookup inconsistencies We varied the accuracy of the failure detectors, and measured the fraction of nodes that have consistent responsibilities. The results are shown in Figure 5.6(a), which illustrates that

lookup inconsistency increases when the failure detector becomes inaccurate, i.e. when it returns more false positives. The plot denoted 'Total inconsistencies' shows the maximum over all possible lookup inconsistencies in a snapshot, whereas 'random lookups' shows the number of consistent lookups when – for each snapshot – lookups are made for 20 random keys, where each lookup is made from 10 randomly chosen nodes. If all lookups for the same key result in the same node, the lookup is counted as consistent. As can be seen, changing the periodic stabilization rate does not effect the lookup inconsistency in this case. This is due to the fact that there is no churn in the system.

Next, we evaluate lookup inconsistencies in the presence of churn. We varied the churn rate with respect to the periodic stabilization (PS) rate of Chord. For the simulations presented here, we choose the inter-arrival time between events of churn (joins/fails) to be one half of the PS delay for 'high churn', the same as the PS delay for 'moderate churn' and 1.5 times the PS rate for 'low churn'. These churn rates correspond to extreme conditions as in reality, the churn rate is very low compared to the PS rate [155]. Figure 5.6(b) shows the results for our experiments when only new nodes join the system. We ignore failures in this experiment as they only affect availability. The y-axis gives a count of the number of lookup inconsistencies per snapshot. As the figure shows, churn does not effect lookup inconsistencies much. In our experiments with a perfect failure detector (probability of false positive = 0), there was non-zero, though extremely low, number of lookup inconsistencies under churn (2.79×10^{-7} for a high churn system).

The reason for lookup consistencies to be almost the same for experiment with and without churn is as follows. An inconsistency in a scenario with perfect failure detectors and churn only happens if multiple nodes join between two old nodes m, n (where $m.succ = n$) before m updates its successor pointer by running PS [47]. The likelihood of this happening in a large system with moderate rate of joins and uniform distribution of node identifiers is low. This effect of node joins on lookup inconsistency can be reduced to zero if we allow lookups to be generated only from nodes that are fully in the system. A node is said to be *fully in the system* after it is accessible from any node that is already in the system. Once a node is fully in the system, it is considered to be in the system until it crashes. For initialization, we define the first node which creates the ring as fully in the system.

Local Responsibilities Next, we evaluate the effect of unreliable failure detectors and churn on consistency dictated by local responsibili-



(a) Without churn (b) With node joins

Figure 5.6: Measurement of lookup inconsistency for various accuracy levels of failure detectors.

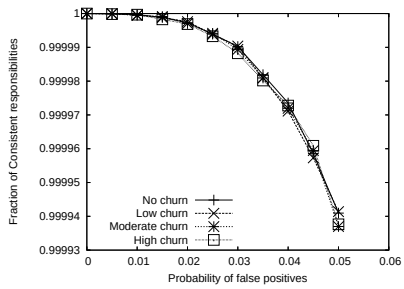


Figure 5.7: Measurement of lookup inconsistencies with respect to local responsibilities, without churn, and with joins-only churn.

ties. Here, we count an inconsistency if the local responsibility (see Section 5.2.1) of multiple nodes overlaps. The results of our simulations are presented in Figure 5.7, which follows our earlier results, showing that the effect of churn on lookup inconsistencies is negligible compared to the accuracy of the failure detectors.

We compare lookup inconsistencies (left y-axis) and lookup inconsistencies with respect to local responsibility (right y-axis) in Figure 5.8. The comparison shows that given a lookup inconsistency, the probability of overlapping responsibilities is approximately only 0.01. This can be seen by the scale of the lookup inconsistency (left y-axis) and lookup inconsistencies due to overlapping local responsibility (right y-axis). The trend of both curves is the same as we reduce the accuracy of the failure detectors (x-axis).

Mutually-dependent Failure Detectors We repeat our previous experiments with mutually dependent failure detectors. In our simulations, if a node n is suspected, the probability of nodes doing accurate detection on n drops to 0.7. This leads to a higher probability for multiple nodes to incorrectly suspect n as failed. In the scenario for the simulations, we suspect 32 random nodes out of 1024 nodes, and do not introduce churn during the experiment. The results are shown in Figure 5.9. While the trend remains the same compared to independent failure detectors, mutually dependent failure detectors produce higher lookup inconsistencies, but still low.

Key Availability Next, we evaluate the percentage of keys available in a system under churn and with inaccurate failure detectors. Experimental studies [130, 57, 155] show that lifetime of nodes staying in a peer-to-peer system ranges from tens of minutes to more than an hour. Further, experiments show that where node's mean lifetime is 1 hour, the optimal freshness threshold for periodic stabilization is approximately 72 seconds [91]. Consequently, for our experiments, we choose a stabilization rate of 1 minute and vary the lifetime of nodes in tens of minutes.

For each snapshot, we count the percentage of keys that are available (see Definition 4). The results of our experiments are shown in Figure 5.10. Availability of keys (the data level) is effected by both inaccuracy of failure detectors and churn. This is different from previous experiments, which showed that lookup inconsistency (the routing level) is effected mainly by the inaccuracy of failure detectors. Even

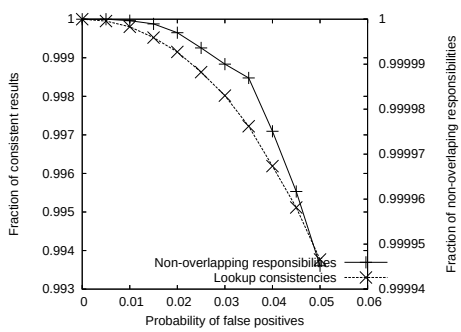


Figure 5.8: Comparison of lookup inconsistency (left y-axis) and lookup inconsistencies with respect to local responsibilities, titled 'Non-overlapping responsibilities' (right y-axis).

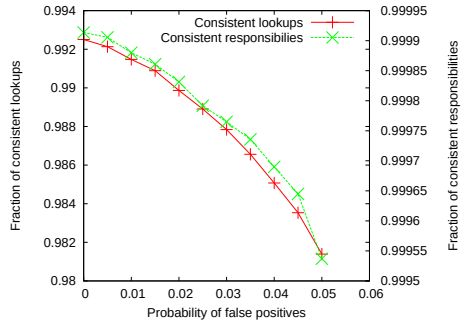


Figure 5.9: Comparison of lookup inconsistency (left y-axis) and lookup inconsistencies with respect to local responsibilities, titled 'Consistent responsibilities' (right y-axis), while using mutually dependent failure detectors.

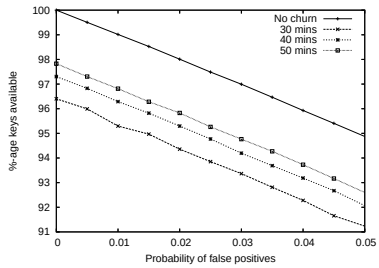


Figure 5.10: Evaluation of percentage of keys available under various levels of churn and inaccurate failure detectors.

with perfect failure detectors (probability of false positive = 0 in Figure 5.10), churn results in unavailability of keys. This is expected as a failure can result in temporary unavailability of keys.

The effect of node joins on key availability can be reduced by using atomic ring maintenance algorithms [40, 49, 126]. These algorithms give a consistent view of the ring in the presence of node joins and leaves by transferring responsibilities of keys before a join or leave completes. Similarly, the effect of node failures on key availability can be reduced by using replication. Such techniques for handling both joins and failures atomically in an overlay are the focus of Chapter 6.

Affect of using Quorums By substituting the probability of an inconsistency, p, in Equation 5.3 with the results of our simulation, shown in Figure 5.7, we get the results plotted in Figure 5.11. The figure shows

the probability for two overlapping majority sets in a certain configuration of an overlay, and its dependence on the probability of false-positives of failure detectors. Our simulation results show that the probability of overlapping majority quorums is high, but declines with degrading failure detectors. Reflecting the results shown in Figure 5.5, the probability that two majority sets are disjoint in a system with an even number of replicas is almost zero. However, lesser unavailable replicas can be tolerated in a system with an even replication degree, since the majority consists of more replicas. The effect of the number of replicas involved in each operation on consistency and performance as been further studied by Bailis *et. al.* [12].

5.4 Discussion

There has been work done on studying lookup inconsistencies under churn. Rhea *et. al.* [125] have explored lookup inconsistencies for Chord. Their approach differs from ours as they define a lookup to be consistent if a majority of nodes concurrently making a lookup for the same key get the same result. For our work, we require all results of making the lookup for the key to be the same. This is an important requirement for guaranteeing data consistency. Furthermore, their work overlooks the fact that imperfect failure detectors mainly cause inconsistent lookups. Zhuang *et. al.* [165] studied various failure detection algorithms in overlay networks. Our work is extended to provide techniques to reduce the effect of lookup consistencies.

Atomic ring maintenance algorithms [40, 49, 126] provide lookup consistency guarantees amid node joins and leaves, ignoring failures and inaccurate failure detectors. As we have shown, the main contribu-

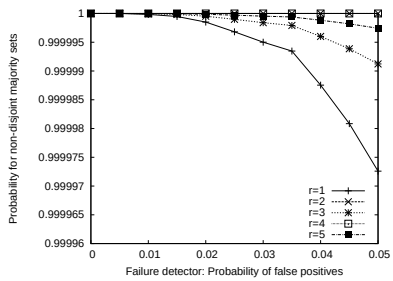


Figure 5.11: Probability of overlap for two majority-based quorum operations in an overlay configuration, for various accuracy levels of failure detectors.

tors to lookup inconsistency are inaccurate failure detectors. Therefore, merely using atomic join and leave operations is insufficient for guaranteeing lookup and data consistency, and choice of a failure detection algorithm is of crucial importance in asynchronous networks.

We have discussed and evaluated two techniques to reduce the affects of lookup inconsistencies: local responsibilities and quorum techniques. While effects of lookup inconsistencies can be reduced by using local responsibilities, we show that using responsibility of keys may affect availability of keys. This is a trade-off between availability and consistency. We have shows that using quorum-based techniques amongst replicas of data items further reduce lookup inconsistencies. Since majority-based quorum techniques require a majority of the replicas to make progress, these algorithms may still make progress even with unavailability of some keys/nodes. Thus, using a combination of local responsibilities and quorum techniques is attractive in scalable applications where consistency is more important than availability.

Data consistency is an important requirement for many applications. Since overlays and DHTs are aimed to operate in asynchronous networks and tolerate node dynamism, guaranteeing data consistency in such systems is non-trivial. The problem of providing data consistency in overlays can be attacked on two levels: routing level and data level. In this chapter, we have focused on the routing level by providing techniques to reduce the effect of lookup inconsistencies, and hence, data inconsistencies. In the following chapter, we focus on the data level. Augmenting to the routing techniques introduced thus far, we build a distributed key-value store that guarantees the strongest form of data consistency for single item operations.

CHAPTER 6

A Linearizable Key-Value Store

Chapter 5 shows that lookup inconsistencies can lead to data inconsistencies. Weaker consistency guarantees, such as eventual consistency [157, 39, 159], suffice for certain applications [38, 81]. Yet, there is a significant class of application that requires stronger consistency guarantees, e.g., financial record applications, services managing critical metadata for large cloud infrastructures [20, 66], or more generally, applications in which the results of data-access operations have external effects. We target such applications, and aim to provide linearizability [63] on top of overlays. In this chapter, we present *CATS*, an elastic and linearizable key-value store that utilizes the properties of overlays, such as incremental scalability and self-management. These properties make CATS an ideal storage system for modern web-scale applications, as they generate and access massive amounts of semi-structured data at very high rates. The work presented in this chapter has been performed in collaboration with Cosmin Arad.

Introduction It has been shown that a web-service can provide only two of the following three properties simultaneously: consistency, availability, and partition tolerance [19, 51]. As any long-lived distributed system will come across network partitions (see Section 3.1), applications generally have to choose between consistency and availability. Section 3.2 and 3.3 present solutions for handling network partitions and

mergers on the routing level. Here, we focus on data level issues as we build CATS on top of Recircle, which takes care of routing level issues.

In Chapter 5, we discuss the trade off where increasing consistency guarantees decreased availability of keys. We show how quorum-based algorithms increase consistency guarantees, yet fall short of providing consistency with probability one. In this chapter, we aim to provide linearizability/atomic consistency (see Section 2.2.2); the strongest form of consistency for single item read and write operations.

Achieving linearizability in static systems Attiya et al. [11] present a quorum-based algorithm, called ABD, to achieve linearizability for a static set of nodes replicating a data item. In ABD, a data item has a sequence/version number associated with it. A write operation fetches the sequence number from a majority of replicas, selects the highest sequence number s, and sends a write message to all replicas with the new value v and sequence number $s + 1$. Upon receiving such a message, a replica updates its locally stored value (and associated sequence number) of the data item if the received sequence number is larger than the locally stored sequence number for that data item. The replica then responds with an acknowledgment message. Ties for sequence numbers are broken using process identifiers. A write operation is considered complete once the writer receives acknowledgments from a majority of replicas. A read operation fetches values with sequence numbers from a majority of replicas, and returns the value with highest sequence number h. To avoid consistency violations due to concurrent incomplete operations, a read operation writes the value read to majority of nodes (using sequence number h) before returning. Using the second phase of write before returning a read is known as *read-impose*. A read-impose is not required if all replies to the read request contain the same sequence number.

Majority-based quorum algorithms block if a majority of the nodes fail, or become unreachable due to network partitions, giving rise to unavailability[1]. A quorum-based algorithm, such as ABD, guarantees linearizability as long as operations overlap on at least one replica.

Problem Statement A naïve approach to achieving linearizability in an overlay is to consider each key-value pair as a data item, and use

[1] Applications that require high availability, and low consistency guarantees, use *hinted-handoff* [38] in such cases. In hinted-handoff, if a replica is unavailable, a temporary replacement replica is created where new writes are stored (thus the term 'hint'). Once the original replica is accessible again, the temporary replica hands over the hints to it.

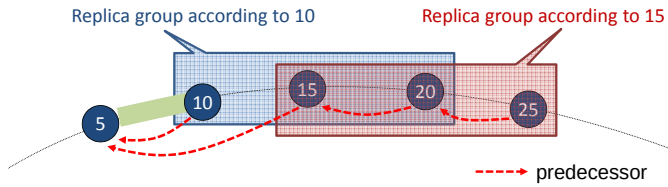


Figure 6.1: Replication groups for key range $k_{(5,10]}$ using SL-replication with replication degree three. Due to inaccuracy of failure detectors, node 15 points to 5 as its predecessor. The replication group for $k_{(5,10]}$ is $\{10, 15, 20\}$ according to node 10, and $\{15, 20, 25\}$ according to node 15. Multiple non-overlapping majority quorums exist, e.g. $\{10, 20\}$ and $\{15, 25\}$.

ABD reads and writes over the replicas of the data item/key-value pair to implement the overlay's $get(k)$ and $put(k, v)$ operations, respectively. Unfortunately, such an approach is incorrect. Chapter 5 discusses how non-overlapping quorums can exist due to routing anomalies in key-based replication. Hence, ABD operations may operate on non-overlapping quorums and violate linearizability. This is not an error of ABD, but due to the breach of the assumption of ABD, i.e. a static set of replicas.

In this chapter, we use SL-replication, which is a node-based replication scheme. We chose SL-replication as it is widely used, and simpler than key-based replication since it does not require bulk operations [40]. Yet, the problem of non-overlapping quorums because of routing anomalies still exists. This is shown in Figure 6.1, where the replication degree is three. Here, due to false failure detection of node 10 by node 15, both 10 and 15 are responsible for the key range $k_{(5,10]}$, i.e., $keys \in (5, 10]$. The replication group for $k_{(5,10]}$ is $\{10, 15, 20\}$ according to node 10, and $\{15, 20, 25\}$ according to node 15. Non-overlapping majorities exist here, e.g. $\{10, 20\}$ and $\{15, 25\}$. This can lead to quorum operations to perform on non-overlapping set of replicas, and thus, violation of data consistency.

6.1 Solution: CATS

In this section, we first provide the intuition and high-level overview of our solution, called CATS, and then, provide its details.

The correctness condition for quorum-based algorithms is that all operations should overlap on at least one node. This is violated in Figure 6.1 because two replication groups exist for $k_{(5,10]}$. To prevent such

an execution, we need to fulfill two requirements: (1) replicas should agree, and know, when the replication group changes, and (2) once the replication group changes, the older replication group should not participate in any quorum operations. Fulfilling requirement 1 ensures that a replica will know whether it is a member of the latest replication group. Fulfilling requirement 2 ensures that quorum operations will only operate on the latest replication group. Satisfying both requirements simultaneously will prevent multiple non-overlapping operable replication groups to exist, thus preventing problematic scenarios that may lead to inconsistencies.

Requirement 1 necessitates achieving *agreement* in a distributed environment, which is what *consensus* algorithms, e.g., Paxos [86], are designed for. Informally, multiple nodes can *propose* different values in an instance of a consensus algorithm. The algorithm ensures that the participating nodes agree on one final decision, which is a value selected from the proposals. In our case, when a replication group changes, the new replication group can be proposed in a consensus instance. Once consensus decides, the next replication group is *installed*.

Requirement 2 demands a mechanism to distinguish between the older and newer replication group. A trivial and common technique for achieving this is to have a version/sequence number associated with a replication group. A newer replication group can then have a higher sequence number than an older replication group. Requirement 2 states that replicas from the older replication group should not participate in quorum operations. Thus, we need a mechanism for a replica to know if it should reply to a quorum operation or not. We achieve this by obliging all quorum operation requests sent to the replicas to include the replication group, and the group's sequence number, that the request is operating on. Upon receiving such a request, a replica should not reply to the request if it (1) is not in the replication group specified in the request, or (2) does not have the same group sequence number as the one specified in the request. Similarly, the quorum operation requester should ignore a reply from a replica if the reply contains a replication group sequence number older than the one the requester is operating on.

Next, we discuss the afore-mentioned techniques in detail, along with algorithmic specification. In our discussions, the *view* of a replication group is represented as $\langle\, k_{(a,b]}, G, i \,\rangle$, where G is the set/group of replica nodes, i is the sequence number, and $k_{(a,b]}$ is the keys being replicated by the view. In short-hand notation, for a view v, v_G denotes the group of replica nodes of v, v_i denotes its sequence number, and v_k

denotes its key range.

6.1.1 Replica Groups Reconfiguration

Algorithms such as ABD are designed for a static set of replicas. Changing the replica set requires a *reconfiguration* operation [88, 99, 26], with special mechanisms to handle on-going read/write operations. In this section, we discuss (1) when and who initiates the reconfiguration, (2) what should be the new replication group upon reconfiguration, (3) how the reconfiguration works, and (4) how the result of the reconfiguration is applied. The reconfiguration algorithms are presented in Algorithms 9, 10 and 11. We discuss the read/write operations in the next section.

Initiator of reconfiguration: Consistent hashing [76] maps responsibility of key ranges to nodes. We use the same principle to decide which node should initiate a reconfiguration operation. A node p is responsible for initiating a reconfiguration if the replication group for a key range in $(p.pred, p]$ has to be changed. As per SL-replication, when a node p joins or fails, it affects r replication groups: the group for $k_{(p.pred,\, p]}$, and $r-1$ replication groups that are the responsibilities of $r-1$ nodes going counter-clockwise from p.

When a new node p joins, the replication group for $k_{(p.pred,\, p]}$ needs to be reconfigured. Since p is responsible for $k_{(p.pred,\, p]}$, it should initiate the reconfiguration operation. For instance, in Figure 6.2(b), node 7 is responsible for initiating the reconfiguration, as the replication group of its responsibility range $(5, 7]$ has to change. Note that the key range responsibility of node 10 is also changing (decreasing). In such cases, to avoid multiple nodes initiating the same reconfiguration, a node initiates the reconfiguration only if it notices an increase in its responsibility range. Thus, node 10 will not initiate the reconfiguration upon 7's join, only 7 will.

In Algorithm 9, a new node joining the system attempts to join the replication group that overlaps with its responsibility. The new node searches for the current replication group of its responsibility range by sending a GetGroupRequest message. Upon receiving a reply (line 16), the new node initiates the reconfiguration to indicate that it wants to join the replication group.

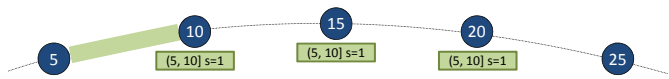
Apart from $k_{(p.pred,\, p]}$, when a node p joins the overlay, it should replicate the responsibilities of $r-1$ nodes going counter-clockwise from p. For instance, if a node 17 joins in Figure 6.2(a), it should replace node 20 as a replica for $k_{(5,\, 10]}$. Since the replication group of node 10's

responsibility has to change, it is responsible for initiating the reconfiguration. Node 10 will discover the presence of 17 when 10's successor-list gets updated (or optionally, 17 can notify 10). At this point, node 10 initiates the reconfiguration to replace the farthest node, 20, in the replication group, with 17. We show such a reconfiguration initiation in Algorithm 9 on lines 25–30.

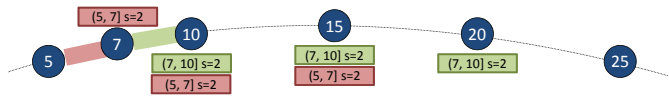
When a node fails, the responsibility of its successor increases (Figure 6.2(c)). The successor of the failed node takes over the responsibility by initiating a reconfiguration to replace the failed node with an alive node in the successor-list. This is shown in Algorithm 10 on lines 7–16. Note that in the absence of checking whether a node's responsibility increased (line 9), the NewPredecessor event would trigger reconfigurations for node joins as well.

A node failure also affects the replication groups of $r-1$ nodes going counter-clockwise. For instance, if node 20 failed in Figure 6.2(a), it should be replaced by node 25 for replication group of $k_{(5,10]}$. For this purpose, each node does failure detection on nodes in the replication group of its responsibility. As soon as the failure detector notifies of a failure (Algorithm 10, line 1), the node initiates a reconfiguration.

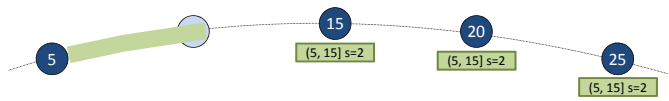
New replication view: Replication groups change when a node joins, fails, or does a false failure detection. When a new node joins, it takes over responsibility for some keys from its successor according to the



(a) A stable configuration showing the replication group for $k_{(5,10]}$.



(b) A joining node 7 breaks the range into two sub-ranges, $k_{(5,7]}$ and $k_{(7,10]}$, thus creating two replication groups.



(c) A failure of node 10 increases the range. Essentially, node 10 is replaced by 25 in the replication group.

Figure 6.2: Reconfiguration for key range $k_{(5,10]}$.

consistent hashing principle. Hence, a join essentially splits a key range, creating two replication groups in-turn. The new views of the replication groups are chosen using the successor-lists and incrementing the sequence number. Consider the overlay configuration and replication group of $k_{(5,10]}$ shown in Figure 6.2(a). The view of the replication group is $v = \langle\ k_{(5,10]}, \{10, 15, 20\}, 1\ \rangle$. Suppose a node with identifier 7 joins the system (Figure 6.2(b)). The new node will break the key range into two key ranges, $k_{(5,7]}$ and $k_{(7,10]}$. Using the SL-replication scheme, and incrementing the sequence number, v is split into two new views: $\langle\ k_{(5,7]}, \{7, 10, 15\}, 2\ \rangle$ and $\langle\ k_{(7,10]}, \{10, 15, 20\}, 2\ \rangle$.

In case of a failure, or false failure detection, the failed/suspected node has to be replaced in the replication group with an alive node. Using the SL-replication scheme, this has to be the next alive node met going clockwise. Starting from the configuration in Figure 6.2(a), the failed node 10 is replaced in the replication group by node 25 in Figure 6.2(c). Thus, the view from Figure 6.2(a) should evolve into the new view $\langle\ k_{(5,10]}, \{15, 20, 25\}, 2\ \rangle$. In case node 10 was falsely detected as failed, it can become part of the replication group again by replacing 25 in the new replication group. This requires a new reconfiguration operation, and is similar to the join process.

In essence, when a replication view changes, it evolves into a set of view(s) of size two if the reconfiguration was triggered by a join, and one if the reconfiguration was triggered by a failure or false failure detection. In a long running system, it may happen that, due to several joins, key ranges for each view become small. Eventually, each view may represent a key range of size one since join operations split a key range. Such a state is completely legal, as each key-value pair is essentially a shared memory register. For reducing the overhead of maintaining a view for each key, consecutive keys can be merged into a single view once the system stabilizes (no churn) for long enough. Such merger of key ranges can be done optimistically (or via a special reconfiguration operation), when replicas notice an opportunity for a merger. We solicit such optimizations to focus on the core problem.

Reconfiguration operation: Changing the set of replicas dynamically, while operations like read and write are occurring, is known as *reconfiguration*. Reconfiguration is a well-known problem as it is required by many online systems, for instance, to replace failed machines, upgrade out-of-date machines, and move replicas from one network location to another. Most of the reconfiguration approaches[2] use consen-

[2] A tutorial on various reconfiguration techniques was written by Aguilera et. al. [5].

Algorithm 9 Reconfiguration (Part 1): Handling joins

Notation: at node n for views stored on n:
$lView \Rightarrow$ view where $n \in view_k$
$aViews \Rightarrow$ all views where $n \in view_s$

```
1:  upon event ⟨ Join | pred, succ ⟩ at n
2:      sendto succ : GetGroupRequest⟨n, pred, n⟩
3:  end event

4:  receipt of GetGroupRequest⟨initiator, begin, end⟩ from m at n
5:      forward := true
6:      for all view ∈ aViews do
7:          if view_k ∩ (begin, end] not ∅ then              ▷ ranges overlap
8:              sendto initiator : GetGroupResponse⟨view⟩
9:              forward := false
10:         end if
11:     end for
12:     if forward = true then
13:         sendto succ : GetGroupRequest⟨initiator, begin, end⟩
14:     end if
15: end event

16: receipt of GetGroupResponse⟨view⟩ from m at n
17:     trigger ⟨ Propose | view : (JOIN, n) ⟩
18: end event

19: upon event ⟨ Decide | view : (JOIN, j) ⟩ at n
20:     if n ∈ view_G then
21:         Install(view, JOIN, j)
22:     end if
23:     CheckIfProposed(view, JOIN, j)
24: end event

        ▷ Check if a new node joined that should be in n's replica group
25: upon event ⟨ SuccessorListUpdated | sucessor-list ⟩ at n
26:     last := LastReplica(lView_G)
27:     if ∃p ∈ sucessor-list such that p ∉ lView_G and
          ModDistance(p, n) < ModDistance(last, n) then
                                  ▷ starting from n, p is closer than last
28:         trigger ⟨ Propose | lView : (REPLACE, last, p) ⟩
29:     end if
30: end event
```

Algorithm 10 Reconfiguration (Part 2): Handling failures

Init: at a node joining a replica view v
installAcks$[v] = \emptyset \Rightarrow$ to gather acks for install
▷ Monitor nodes in the replica group of $(pred, n]$
1: **upon event** $\langle$ Failed $\mid f \rangle$ **at** n
2: **if** $f \in lView_G$ **then**
3: $alive :=$ ReplaceWith(f) ▷ Next alive node in successor-list
4: **trigger** $\langle$ Propose $\mid lView : (\text{REPLACE}, f, alive) \rangle$
5: **end if**
6: **end event**
▷ An increase in responsibility implies failure of predecessor
7: **upon event** $\langle$ NewPredecessor $\mid oldPred, newPred \rangle$ **at** n
8: $alive :=$ ReplaceWith(f) ▷ next alive node in successor-list
9: **if** $oldPred \in (newPred, n]$ **then** ▷ responsibility increased
10: **for all** $v \in aViews$ **do**
11: **if** $v_k \cap (newPred, oldPred]$ **not** $\emptyset$ **then** ▷ ranges overlap
12: **trigger** $\langle$ Propose $\mid v : (\text{REPLACE}, f, alive) \rangle$
13: **end if**
14: **end for**
15: **end if**
16: **end event**

17: **upon event** $\langle$ Decide $\mid view, (\text{REPLACE}, f, a) \rangle$ **at** n
18: Install$(view, \text{REPLACE}, f, a)$
19: CheckIfProposed$(view, \text{REPLACE}, f, a)$
20: **end event**

21: **receipt of** Install$\langle view \rangle$ **from** m **at** n
22: installAcks$[view]$:= installAcks$[view] \cup \{m\}$
23: **if** $\mid$ installAcks$[view] \mid = \lfloor \frac{r}{2} \rfloor + 1$ **then** ▷ acks from majority
24: **if** $n \in view_k$ **then**
25: $lView := lView \cup view$
26: **else**
27: $aViews := aViews \cup view$
28: **end if**
29: DataTransfer$(view)$
30: **end if**
31: **end event**

▷ Check if n is not in the replica group of its responsibility
32: **every** γ **time units at** n
33: **if** $(pred, n] \notin lView_k$ **then**
34: **sendto** succ : GetGroupRequest$\langle n, pred, n \rangle$ ▷ periodically retry
35: **end if**
36: **end event**

Algorithm 11 Reconfiguration (Part 3): Install

```
1:  procedure Install(view : (JOIN, j)) at n
2:      (start, end] := view_k                                  ▷ split key range
3:      v1_k := (start, j]
4:      v2_k := (j, end]
5:      v1_i := view_i + 1                                      ▷ increment version number
6:      v2_i := view_i + 1
7:      last := LastReplica(view_G)
8:      v1_G = view_G − {last} ∪ {j}                            ▷ create new groups
9:      v2_G = view_G
10:
11:     lView := lView − view                                   ▷ remove old configuration
12:     aViews := aViews − view
13:     if n = last then                                        ▷ add v1 if needed
14:         MarkDataForDeletion(v1_k)                           ▷ delete after sent to j
15:     else
16:         aViews := aViews ∪ v1
17:     end if
18:     if n ∈ v2_k then                                        ▷ add v2
19:         lView := lView ∪ v2
20:     else
21:         aViews := aViews ∪ v2
22:     end if
23:     sendto j : Install⟨v1⟩
24: end procedure

25: procedure Install(view, (REPLACE, f, a)) at n
26:     v := view
27:     v_i := v_i + 1                                          ▷ increment version number
28:     v_G := v_G − {f} ∪ {a}                                  ▷ replace node in group
29:     lView := lView − view                                   ▷ remove old configuration
30:     aViews := aViews − view
31:     if n = f then                          ▷ current node should leave the group
32:         MarkDataForDeletion(view_k)                         ▷ delete after sent to a
33:     else                                                    ▷ add back new view v
34:         if n ∈ v_k then
35:             lView := lView ∪ v
36:         else
37:             aViews := aViews ∪ v
38:         end if
39:     end if
40:     sendto a : Install⟨v⟩
41: end procedure
```

sus[3], e.g. [88, 99, 26, 143]. We denote a reconfiguration request of view v by $(v : \mathsf{O})$, where the operation O can be either:

- $\langle \mathsf{JOIN}, j \rangle$ for a node j joining v such that $j \in v_k$, or
- $\langle \mathsf{REPLACE}, f, a \rangle$ where node $f \in v_G$ has to be replaced by a node a, e.g., due to failure of f.

For $(v : \mathsf{O})$, the operation O is proposed via consensus in our solution. To distinguish between various instances of consensus, we use the old view v to denote the instance of consensus. The nodes in the old view v_G act as the acceptors and decide on the operation that leads to the next view. For *liveness* [82], a majority of nodes in the old view should be alive for consensus to terminate. The nodes in the old view also act as learners for the outcome of consensus. When the nodes in the old view decide, they learn about the decision and move to the new view by applying the decided operation; a process called *installing* the new view (discussed in the next section). Thereafter, the next view is installed on the node that became part of the replication group, i.e., j for $\langle \mathsf{JOIN}, j \rangle$, and a for $\langle \mathsf{REPLACE}, f, a \rangle$. Here, a majority of nodes in a view should be accessible for the view to be reconfigured. Note that such an approach of reconfiguration fulfills our Requirement 1 because nodes in the old view act as learners, and install the new view.

The reason consensus is required for reconfiguration is that under high churn, it may happen that two different next view reconfigurations are proposed concurrently. For instance, consider the replication group $\{10, 15, 20\}$ for keys $k_{(5,10]}$ in Figure 6.2(a). If node 7 joined and node 20 failed simultaneously, two different evolution of the replication group will be initiated: 7 will attempt to create a replication group $G_1 = \{7, 10, 15\}$ for $k_{(5,7]}$, while for the same keys, 10 will attempt to evolve the replication group into $G_2 = \{10, 15, 25\}$ as it maybe unaware of 7's join. Such evolutions will result in non-overlapping quorums on G_1 and G_2: quorum $\{7, 10\}$ in G_1 and quorum $\{15, 25\}$ in G_2, potentially leading to violation of linearizability. Using consensus, only one of the two reconfiguration attempts will succeed as a consensus instance decides on one value.

Using consensus ensures that nodes in the old view v agree on the next view v', and will install the same new view. When a reconfiguration proposer p finds out that the operation decided in instance v is different than what it proposed, it evaluates whether its reconfiguration proposal is still needed. If so, it builds a new reconfiguration

[3]Dynastore [6] is one system that does not use consensus for reconfiguring an atomic register.

operation, reflecting the new view v'. Node p then proposes in consensus instance v', thus evolving the view sequentially (one change at a time). For instance, in the afore-mentioned example where $(v : \langle \mathsf{JOIN}, 7\rangle)$ and $(v : \langle \mathsf{REPLACE}, 20, 25\rangle)$ are proposed simultaneously, assume $\langle \mathsf{REPLACE}, f, a\rangle$ was decided in the consensus instance v. Upon installation, the nodes in v_G will move to view $v' = \langle\ k_{(5,10]}, \{10, 15, 25\}, 2\ \rangle$. When node 7 learns that its proposal was not chosen in instance v, it notices that it still needs to join the replication group. Hence, node 7 initiates $(v' : \langle \mathsf{JOIN}, 7\rangle)$.

In Algorithm 9 and 10, we use consensus as a black-box. The consensus abstraction uses two events, Propose and Decide. As noted earlier, we use a replication group's view as the consensus instance identifier. To propose a value p in a consensus instance v, a node triggers an event $\langle\ \mathsf{Propose} \mid v : p\rangle$. Here, the nodes v_G act as the *acceptors* for instance v. Once a value d is decided for a consensus instance v, a node receives the event $\langle\ \mathsf{Decide} \mid v : d\rangle$. We assume that the proposer and acceptors are the *learners* as well, i.e., the Decide event for instance v is triggered on all nodes in v_G and the proposer. If an acceptor receives a consensus request for an instance that had already been decided, it can reply to the proposer with the decision for that instance. Similarly, if an acceptor in view $v1$ receives a consensus request for instance $v2$ such that $v2_i > v1_i$, it does not participate in the consensus algorithm until it has installed view $v2$. This is required to fulfill Requirement 2.

Upon receiving the Decide event, we use the method CheckIfProposed(v, *decision*) (line 23 in Algorithm 9, and line 19 in Algorithm 10) to check if n had proposed a reconfiguration p for *view*, but the decision is not p. If not, the call should evaluate if the node's reconfiguration is still needed, and propose another reconfiguration operation if so.

Configuration installation: Once a reconfiguration $(v : \mathsf{O})$ is decided, nodes $n \in v_G$ install/apply the operation O. Applying an operation on the view is done as discussed in 6.1.1, and outlined in Algorithm 11. Say applying $(v : \mathsf{O})$ leads to view v'. The installation means that nodes v_G locally store v', thus promising that they will not participate in any quorum operation (read/write) request that does not contain v'. Similarly, after installing v', a node always attaches v' with any responses to quorum requests. A node n, where $n \in v_G$ but $n \notin v'_G$, is no longer in the replication group, and thus, does not reply to any requests for the key range v'_k. Installing a view on a node is a promise that the node will obey Requirement 2.

Once a majority of nodes in v_G install the new view(s) for $k_{(a,b]}$, the

installation is done at the new node that became a replica of the key range. The new node p fetches data from nodes in v_G, and can start replying to requests for the key range once it has received the data. To maintain linearizability when the new replica p fetches the data, p should perform an ABD read on nodes v_G for all keys v_k. This means that for each key, p should get the value with the highest time stamp from a majority in v_G. Otherwise, linearizability would be violated as follows. Consider a key $k = \langle A, 1\rangle$, where A is the value for k, and 1 is k's ABD time stamp. Say $v_G = \{l, m, n\}$, and node p is replacing node n in the view. Assume a $put(k, B)$ terminates on v_G by updating majority $\{m, n\}$, i.e., the local values stored at m and n are now $k = \langle B, 2\rangle$. Instead of the data transfer performing an ABD read on v_G, if p transfered the data from only one replica, it may get the older value $k = \langle A, 1\rangle$ from node l. After installation, a majority ($\{m, p\}$) exists with the older value A at the nodes in the new view $v'_G = \{l, m, p\}$. An ABD read may thus return the older value A, hence violating linearizability. Therefore, to transfer data, the new node p should perform an ABD read for all keys v_k.

To optimize the data transfer, we use a bulk transfer for a key range. Before transferring values, each replica first transfers keys and their timestamps to the new node. Based on the timestamps, the new node retrieves the latest value from the node with the highest timestamp for each key. This avoids unnecessary transfers of values from existing replicas to the new replica, thus lowering bandwidth usage. Furthermore, the new node transfers data in parallel from existing replicas, by splitting the requested data among all replicas. This results in better bandwidth utilization, fast data transfer due to parallel downloads, and avoids loading a single replica. If a replica fails during data transfer, the requesting node reassigns the requests sent to the failed node to the remaining alive replicas.

6.1.2 Put/Get Operations

Operations in quorum-based algorithms like ABD assume a static set of replicas. To emulate such a static set in a dynamic environment, we need to cater for two cases. First, if the set of replicas does not change during an operation (i.e., all requests of the operation operate on the same view), the operation should succeed. Second, if the set of replicas changes during an operation (i.e., some requests of the operation operated on the old view, while some operated on the new view), the operation should detect such cases, abort, and retry the operation from scratch.

The critical correctness condition is to ensure that all requests of an operation occur on the same view. Our solution to achieve this condition is to use *consistent quorums*.

Definition 1. *For a given replication group G, a quorum Q is a* ***consistent quorum*** *of G if all nodes in Q have the same view v of G when the quorum is assembled.*

In a static system, all quorums are consistent quorums since the view does not change. In a dynamic system, to ensure that a quorum is a consistent quorum, when an algorithm assembles a quorum, it only counts replies/acknowledgments from replicas that are in the same view. To accomplish this, a replica includes its view of the replication group with every response/acknowledgment of a quorum request. If a consistent quorum cannot be assembled, it implies that the replication group changed during the operation.

Certain quorum-based algorithms, such as ABD, use two phases. To emulate a static system over a dynamic environment, the view of the consistent quorum of the first phase should be the same as the view of the consistent quorum of the second phase. To achieve this, when the operation initiator assembles a consistent quorum with view v in the first phase, it includes v with all requests in the second phase to replicas in v. Upon receiving such a request, a replica replies (acknowledges) only if its view of the replication group is the same as the view in the request. If the view of the replication group was reconfigured between the first and second phases of a put/get operation, the consistent quorum from first phase cannot be assembled in the second phase, and thus, the operation has to be retried from scratch.

In our solution, we modify the *put* and *get* operations of ABD to use consistent quorums. Next, we present our solution, along with algorithmic details.

The node carrying out the ABD protocol with the replicas is called the *coordinator* node. Algorithm 12 shows the interface of the coordinator to the client. Upon receiving a request for key k from the client, the coordinator starts the ABD algorithm by sending ReadA messages to the replicas of k. The replicas can be found either via lookups, or locally if the coordinator has the information cached. Such information about the replicas is allowed to have lookup inconsistencies. Furthermore, our put/get algorithms are independent of the replication scheme, and how the replicas are found.

Algorithms 13 and 14 show our modified ABD algorithm augmented with consistent quorums. At the core of the algorithm is our method,

$v :=$ ConsistentQuorum(V), where V is a list of views. If a consistent quorum exists in V, the method returns the view v of the consistent quorum. This implies that the number of views in V that are equal to v is at least $\lfloor \frac{r}{2} \rfloor + 1$, i.e. a majority, where r is the replication degree. If a consistent quorum does not exist, the method returns a $nil(\perp)$ value. Compared to stock ABD, whenever the ABD client assembles a quorum, we make sure that the quorum is in fact a consistent quorum. This is done in Algorithm 13 on lines 3-4 and 17-18.

As noted earlier, when a replica receives a quorum request in the second phase, it has to ensure that the replication group has not changed since the first phase. This is accomplished in Algorithm 14 on line 5, where the replica compares its local view to the view included in the request. The replica only replies if the two views are the same.

Algorithm 12 Operation coordinator (Part 1): Client interface

Init: for all keys k:
writeValue[k]:=$\perp$, reading[k]:=$false$, client[k]:=$\perp$

1: **receipt of** GetRequest$\langle k \rangle$ **from** *Client* **at** n
2: reading[k]:=*true*
3: client[k]:=*Client*
4: *replicas* := GetReplicas(k)
5: **sendto** $\forall p \in replicas$: ReadA$\langle$k$\rangle$
6: **end event**

7: **receipt of** PutRequest$\langle k, v \rangle$ **from** *Client* **at** n
8: reading[k]:=$false$
9: client[k]:=*Client*
10: writeValue[k] := v
11: *replicas* := GetReplicas(k)
12: **sendto** $\forall p \in replicas$: ReadA$\langle$k$\rangle$
13: **end event**

14: **upon event** $\langle$ OperationResponse $|$ k, v $\rangle$ **at** n
15: **if** reading[k]:=*true* **then**
16: **sendto** client[k] : GetResponse$\langle k, v \rangle$
17: **else**
18: **sendto** client[k] : PutResponse$\langle k \rangle$
19: **end if**
20: **end event**

To keep the algorithmic specification simple, we omit details about negative acknowledgments (NACK). Here, if a replica is in a different

Algorithm 13 Operation coordinator (Part 2): Actions as ABD client

```
    Init: for all keys k:
          readAcks[k]:=∅, writeAcks[k]:=∅
          readValue[k]:=⊥, group[k]:=∅

 1: receipt of ReadB⟨k, timestamp, value, view⟩ from Replica at n
 2:     readAcks[k] := readAcks[k] ∪ {(timestamp, value, view)}
 3:     group[k] := ConsistentQuorum(readAcks[k])
 4:     if group[k] not ⊥ then                    ▷ if consistent quorum exists
 5:         (t, v, v_w) := HighestTimestampValue(readAcks[k])
 6:         if reading[k] = true then
 7:             readValue[k] := v
 8:             sendto ∀p ∈group[k] : WriteA⟨k, t, v, group[k]⟩
 9:         else
10:             sendto ∀p ∈group[k] : WriteA⟨k, t + 1, v, group[k]⟩
11:         end if
12:         readAcks[k]:=∅                                  ▷ reset ACKs
13:     end if
14: end event

15: receipt of WriteB⟨k, view⟩ from Replica at n
16:     writeAcks[k] := writeAcks[k] ∪ {view}
17:     group[k] := ConsistentQuorum(writeAcks[k])
18:     if group[k] not ⊥ then
19:         trigger ⟨ OperationResponse | k, readValue[k] ⟩
20:         writeAcks[k]:=∅                             ▷ reset ACKs
21:     end if
22:     ResetLocalState(k)                          ▷ as done in Init
23: end event
```

view than the view attached in the request, it can send back a NACK. Such NACKs do not count towards assembling a quorum. When a node receives NACKs and cannot assemble a consistent quorum, it needs start the operation again.

6.1.3 Network Partitions and Mergers

An underlying network partition can cause nodes to be split into disconnected components, where nodes in one component cannot communicate with nodes in another component. Consequently, a network partition results in false failure detections, where nodes in one component detect nodes in the other component to have failed.

Algorithm 14 Replication group member

Init: for all keys k:
version$[k]$:=0, value$[k]$:=$\perp$
view $\leftarrow$ (As assigned by reconfiguration protocol)

1: **receipt of** ReadA$\langle k \rangle$ **from** *Coordinator* **at** *Replica*
2: **sendto** *Coordinator* : ReadB$\langle k, version[\text{k}], value[\text{k}], view \rangle$
3: **end event**

4: **receipt of** WriteA$\langle k, timestamp, value_n, view_{consistent} \rangle$ **from** *Cor* **at** *R*
5: **if** $timestamp >$ version$[k]$ **and** $view_{consistent}$ = view **then**
6: value$[k]$:= $value_n$ ▷ update local copy
7: version$[k]$:= $timestamp$
8: **sendto** *Cor* : WriteB$\langle k, view \rangle$
9: **end if**
10: ▷ else, send a NACK
11: **end event**

On the routing level, Recircle ensures that each component converges into its own overlay. This results in updates of successor and predecessor pointers, such that overlays in each component cover the whole identifier space (shown in Figure 6.3). Therefore, node responsibilities overlap across components. Using stock ABD read and write operations would succeed in each component for the same key, thus violating linearizability. On the other hand, consistent quorums do not exist for the same key in different components simultaneously. Consistent quorums only exist for view $v = \langle\ k_{(a,b]},\ G,\ i\ \rangle$ in a partitioned component $\mathcal{C}$ if a majority of v_G is in $\mathcal{C}$. Since a majority can only be accessible in at most one component, consistent quorums may only exist in one component.

In each component, inaccessible replicas should be replaced by alive nodes to preserve the replication degree. Thus, replication views need to evolve from the views prior to the partition. As inaccessible nodes are detected as failures, the evolution of views is triggered by our reconfiguration algorithms.

Quorum-based algorithms, such as Paxos and ABD, make progress as long as a quorum (majority in our case) is reachable. When a partition occurs, nodes issue reconfiguration operations to replace inaccessible nodes. A reconfiguration operation succeeds only if a majority of replicas in the current view are accessible. Since only one partition can contain a majority for a view, reconfiguration for a view will only suc-

ceed in one component. This preserves our invariant that views evolve sequentially, and multiple non-overlapping quorums do not exist.

In Figure 6.3, the reconfiguration for $k_{(5,10]}$ will succeed in component $\mathcal{O}$ (Figure 6.3(c)), while any reconfiguration for $k_{(5,10]}$ will fail in component $\mathcal{E}$ (Figure 6.3(b)) as the majority in $k_{(5,10]}$'s current view before the partition, $\{10, 15, 25\}$, is not accessible in component $\mathcal{E}$. Eventually, key ranges that have a majority of nodes accessible will evolve into replication groups where all replicas are alive and accessible. In other components, reconfiguration operations for the same key ranges will be tried but will fail.

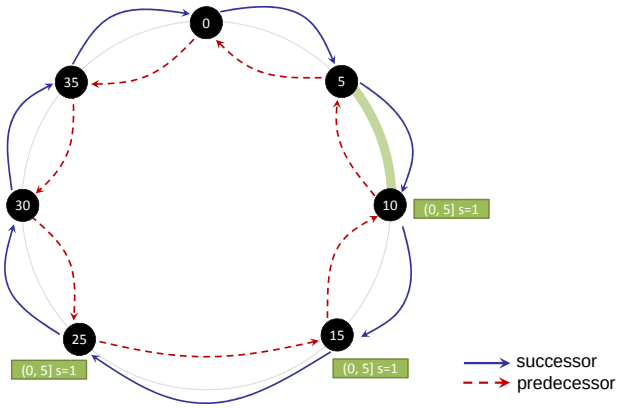
Even if a reconfiguration operation fails in a component, the node initiating the operation should retry it periodically. As we discuss in the next section, periodic retries are required once the network partition ceases and the components merge.

Since ABD is a quorum-based algorithm, its operations succeed only if a quorum is accessible. Thus, operations for a key k succeed in components where a majority, using consistent quorums, of replicas for k is accessible. The keys become unavailable in the components where a majority of replicas is inaccessible. For instance, put/get operations for key 7 will fail in component $\mathcal{E}$ in Figure 6.3(b); thus, key 7 is unavailable in that partition. On the other hand, operations for key 7 will succeed in component $\mathcal{O}$ (Figure 6.3(c)) as a majority of nodes from 7's replication group before the partition exits in $\mathcal{O}$. This tradeoff between availability and consistency amid network partitions is in line with the design space of CATS.

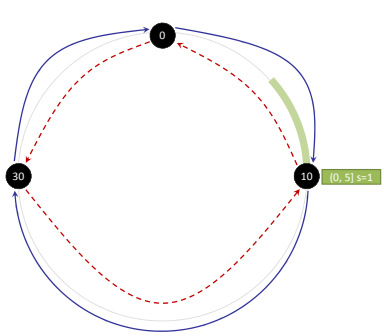
Network Mergers

An underlying network merger results in the disconnected components to be able to communicate again. On the routing level, Recircle guarantees that the routing pointers, including successor and predecessor pointers, will be fixed to depict a single ring. The updated successor and predecessor pointers trigger reconfigurations operations to reflect the replication groups on the routing level, dictated by successor-list replication, on the data level.

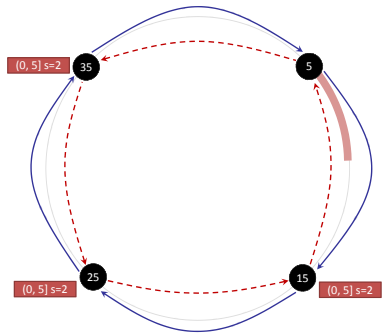
If a network partitions into more than $\frac{r}{2}$ components, where r is the replication degree, it may happen that none of the components contains a majority of replicas for a key range. For instance, a network may partition into r components such that each component contains only one replica. The view for such a key range cannot change as any reconfiguration request for the key range will fail. Periodically retrying failed reconfiguration operations ensures that once the network merges, or



(a) A stable overlay showing the replication group for $k_{(5,10]}$.



(b) A partition $\mathcal{E}$ containing nodes $\{0, 10, 30\}$.



(c) A partition $\mathcal{O}$ containing nodes $\{5, 15, 25, 35\}$.

Figure 6.3: Reconfiguration for key range $k_{(5,10]}$ before and after a network partition. Figure (a) shows the original overlay before the network partition. Figures (b) and (c) show the independent overlays when the underlying network partition splits the nodes into two components: (b) $\mathcal{E} = \{0, 10, 30\}$, and (c) $\mathcal{O} = \{5, 15, 25, 35\}$. $\mathcal{E}$ does not contain a majority of nodes from the replication group for $k_{(5,10]}$ before the partition, hence, the view for $k_{(5,10]}$ cannot be reconfigured in this partition. The view for $k_{(5,10]}$ can be reconfigured in $\mathcal{O}$ as it contains a majority from the view before the partition.

the network partially merges such that some of the components can communicate, the reconfiguration will succeed.

Retrying a failed reconfiguration operation is also necessary to handle the following case. Due to a partition, the replicas may get divided into components such that the node with the lowest identifier in the replication group gets evicted. For instance, consider the replication group $k_{(5,10]}$ with view $v = \langle\ k_{(5,10]}, \{10, 15, 25\}, 1\ \rangle$ in Figure 6.3(a). Due to a network failure, node 10 gets partitioned into component $\mathcal{E}$, while nodes 15 and 25 into component $\mathcal{O}$. Since the responsibility range increases for node 15 and it detects that 10 has failed (as 10 is no longer accessible), node 15 will initiate a reconfiguration operation to replace 10, with the next alive node 35 in its successor-list. Such a reconfiguration will succeed as a majority of replicas from v, $\{15, 25\}$, are accessible. The result of the successful reconfiguration is shown in Figure 6.3(c), where $s = 2$. On the other hand, when node 10 will detect the failure of 15 and 25, its reconfiguration attempt will fail, and the view will remain stuck in 10's component (Figure 6.3(b)). In such cases, node 10 should retry the reconfiguration periodically since after merger, it is the responsible node for $k_{(5,10]}$ as per consistent hashing. Hence, the retrial will make sure that 10 will take over responsibility of $k_{(5,10]}$ after merger. Note that a node n initiates reconfigurations for only key range $(n.pred, n]$, thus each node keeps vying to be included in the replication group of its responsibility as dictated by consistent hashing.

False failure suspicions are similar to transient network partitions and mergers. A false failure suspicion can lead to an alive node being evicted from a replication group, which is similar to a node eviction due to a partition. A reconfiguration due to a false failure suspicion is rectified in a similar manner as when a network partition heals, where the responsible node attempts to become part of the replication group. This can be seen in Figure 6.2(c), where node 10 may have been removed from the replication group due to a false suspicion by node 15.

Crash-recovery, while using persistent storage, is akin to the node being partitioned away for a while. In both cases, when a node recovers or the partition heals, the node has the same view and data as it had before the failure/partition. Hence, our algorithms already support crash-recovery since they are partition tolerant.

Garbage Collection

If a network remains partitioned for a long duration, replicas may exist that are no longer needed. Consider a replication group G for $k_{(a,b]}$, and

a node $n \in G$, where n is the furthest replica as per SL-replication[4]. If n gets partitioned away, G may still have a majority quorum in the other partition $\mathcal{P}$ (such that $n \notin \mathcal{P}$). G can then be reconfigured, and thus evolve into subsequent new groups. While the network is partitioned, new nodes may join partition $\mathcal{P}$. Depending on their identifiers, such new nodes can become part of G. Assume a node m joins $\mathcal{P}$ such that the identifier m is closer to $k_{(a,b]}$ clockwise than n. Therefore, even after the merger, n would not become part of the replication group G. The data on n for $k_{(a,b]}$ is thus unnecessary. Such useless copies of data should be detected and removed.

A garbage collection mechanism is required to avoid unnecessary copies of data lingering around in the system as a result of transient network partitions. We employ a periodic garbage collector for this purpose. The garbage collector identifies useless copies and removes them. Consider a view $v = \langle\, k_{(a,b]},\, G,\, i \,\rangle$. For a node n, in view v, to identify if it is storing an unnecessary copy for $k_{(a,b]}$, n periodically searches for replicas of $k_{(a,b]}$ by making lookups for keys a and b. A node initiates such searches only if notices that a majority in v_G has failed. As a result of the search, if n finds a node with view $v' = \langle\, k_{(a,b]},\, H,\, j \,\rangle$, such that $n \notin v'_G$ and $v'_j > v_i$, it implies that v has evolved/been reconfigured. To ensure availability of data, n contacts all nodes in v'_G to confirm that they have the data for $k_{(a,b]}$. Upon receiving acknowledgments from a majority in v'_G, n can safely remove the data.

A network partition can lead to data copies becoming unnecessary. Nevertheless, in a distributed asynchronous environment, a node cannot be sure if its copy is unnecessary until the partition has ceased. Thus, even if a copy is unnecessary given an oracle/global view, our periodic garbage collector will not remove any data unless it can reach a majority in the newer view. In this way, we distinguish between a group that has evolved during a partition (data can be garbage collected) than one where neither partition contained a majority (the data is essential).

6.1.4 Correctness

Linearizability is violated if two operations use non-overlapping quorums. For a static system, correctness can be proved by showing that quorums for all operations always (from time zero, till eternity) overlap. For a dynamic system such as CATS, correctness can be proven by showing that (1) at any time, any two quorums will overlap, and (2) linearizability is preserved when data is copied to a new replica. Our

[4] e.g., node 25 in the replication group of $k_{(5,10]}$ in Figure 6.3(a)

mechanism of data transfer ensures that linearizability is preserved. Thus, to prove correctness, we have to show that for any key range, two non-overlapping quorums do not exist simultaneously. In other words, at any point in time, consistent quorums will overlap.

In this section, we consider a key range $k_{(a,b]}$ in view $v = \langle\, k_{(a,b]}, \text{G}, \text{i} \,\rangle$, that reconfigures into view $v' = \langle\, k_{(a,b]}, \text{H}, \text{i+1} \,\rangle$. At any time, the replication group can be in three states: (1) before reconfiguration, (2) during reconfiguration, and (3) after reconfiguration. Since the replication degree r is constant, and we reconfigure one change at a time, there may exist only one node $n \in v_G$ such that $n \notin v'_H$, and there exists only one node $m \in v'_H$ such that $m \notin v_G$.

Case 1: Before reconfiguration, i.e., in v, a consistent quorum is similar to a regular quorum. Any consistent quorum will contain a majority from v_G. From the majority principle, we know that any two majorities from v_G will overlap on at least one node. Hence, any two consistent quorums will always overlap before reconfiguration, where all replicas are in view v.

Case 2: A reconfiguration is in-progress until a majority in v_G has installed (moved to) v'. Here, m cannot install (become part of) v' until a majority in v_G has installed (moved to) v'. During reconfiguration, some nodes from v_G are in view v, while some are in v'. If a majority is still in v_G, any two consistent quorums for v will overlap, while a consistent quorum does not exist for v' since only a minority from v_G is in v'. Once a majority from v_G has moved to v', no consistent quorum exists for v since only a minority of v_G is in view v. Here, any two majority consistent quorum in v' will overlap.

Some quorum algorithms, such as ABD, have two phases. It may happen that in the first phase, an operation gets a consistent quorum for v, yet a majority has moved to v' before the second phase. In such a case, the operation will fail in the second phase since it will attempt to gather a consistent quorum for v, which does not exist anymore. Failing such an operation is required to give a quorum-based algorithm the impression that it is operating on a static set of nodes [5].

Disjoint majorities may exist if the number of active replicas exceeds the replication degree r. To ensure that only r replicas exist at any point, the new replica group member m does not become part of v' until v' has been installed on a majority in v_G. Once v' is installed on a majority of

[5] Due to this property, we believe the idea of consistent quorums can be used in other quorum-based algorithms.

nodes in v_G, it implies that the outgoing node n can no longer be part of any consistent quorum. Say v_{maj} is the majority in v_G that installed and moved to view v'. If $n \in v_{maj}$, then upon installing v', n will notice that it is no longer part of the replication group. Hence, it will not reply to in any quorum operation requests. If $n \notin v_{maj}$, then n will attach view v with its replies to quorum operation requests. Yet, a consistent quorum for v cannot be assembled because a majority in v_G has already installed v'. Hence, n can no longer be part of any consistent quorum once a majority in v installs the new view v'. At this point, the number of nodes that can be part of a consistent quorum for view v' is $r - 1$. Thus, it is safe for the incoming node m to install v', and reply to quorum requests with v'.

Case 3: Once the reconfiguration terminates, a minority of nodes can be in v and a majority is in v'. There on wards, it is impossible to assemble a consistent quorum with view v. Since $\mid v'_H \mid \leq r$, any two majority consistent quorums with view v' will overlap. The number of nodes in v' equals r as soon as all nodes in v'_H have installed the new view v'.

If a consistent quorum exists for a view $\langle$ k, X, j $\rangle$, then no consistent quorum can exist for any view for keys k with a sequence number less than j. For instance, assume reconfiguration r_1 evolves view v into view v'. Thereafter, another reconfiguration r_2 happens which changes view v' to v''. The completion of r_1 ensures that only a minority of nodes are in view v. Similarly, completion of r_2 ensures that only a minority of nodes are in view v'. Hence, once v'' is installed on a majority of nodes, no consistent quorum can exist for any view with sequence number than v''. For example, there cannot be a consistent quorum with view v.

6.2 Evaluation

In this section, we evaluate the performance, in terms of throughput and operation latencies, as well as the scalability and elasticity of our implementation of the CATS system. Furthermore, we evaluate the performance overhead of achieving consistency, and we perform a comparison with Cassandra 1.1.0, a system with an architecture similar to CATS. CATS was implemented (in Java) using the Kompics message-passing component framework [10], which allows the system to readily leverage multi-core hardware by executing concurrent components in parallel on different cores. To avoid being side-tracked, and for efficient

lookups (resolved in $O(1)$ hops [60]), we maintained a full-view at each node, e.g., by using the Cyclon [162] peer-sampling service.

We ran our experiments on the Rackspace cloud infrastructure, using 16 GB RAM server instances. We used the YCSB [30] benchmark as a load generator for our experiments. We evaluated two workloads with uniform distribution of keys; a read-intensive workload comprising of 95% reads and 5% updates, and an update-intensive workload comprising of 50% reads and 50% updates. We chose to perform updates instead of inserts in the workload to keep the data set constant throughout the experiment. Such a choice does not have side-effects since the protocol for an update operation is the same as the one for an insert operation. Unless otherwise specified, we used data values of size 1 KB. We placed the servers at equal distance on the consistent hashing ring to avoid being side-tracked by load-balancing issues. CATS supports persistent storage through the Java Edition of BerkeleyDB (SleepyCat) [14], LevelDB [46], and Blsm (blsm). For our experiments reported here, we used an in-memory sorted map.

6.2.1 Performance

In the first set of experiments, we measured the performance of CATS in terms of the average latency per operation and the throughput of the system. We increased the load, i.e. the dataset size and the operations request rate, proportionally to the number of servers, by increasing the number of keys initially inserted into CATS, and the number of YCSB clients, respectively. For instance, we load 300 thousand keys and use 1 client for generating requests for 3 servers, 600 thousand keys and 2 clients for 6 servers, and so on. For each system size, we varied the request load by varying the number of threads in the YCSB clients. For low number of client threads, the request rate is low and thus the servers may be under-utilized, while a high number of client threads can overload the servers. We started with 4 threads, and doubled the thread count each time for the next experiment until 128 threads.

Figure 6.4 shows the results, averaged over three runs, for various number of servers. For each server count, as the request load increases, the throughput also increases till a certain value after which, only latency increases without an increase in throughput. Such a state depicts that the system is saturated and cannot offer more throughput. In other words, when the system is underloaded (few client threads), the latencies are low yet server resources are not fully utilized. As the request rate is increased by increasing the number of client threads, the latency and throughput also increase until a certain throughput is offered. For

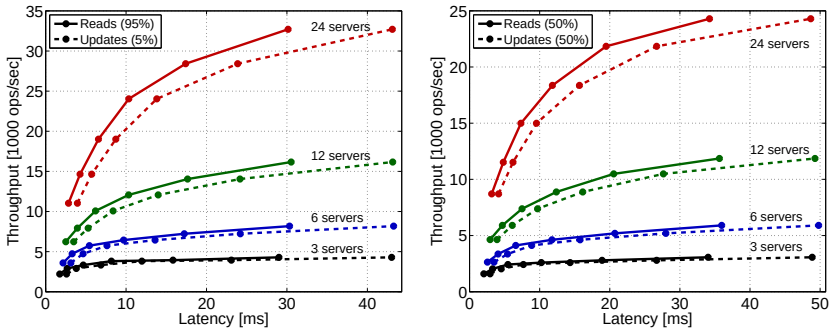


Figure 6.4: Performance for a read-intensive (left) and an update-intensive (right) workload.

instance, for 3 servers and a read-intensive workload, the system saturates at approximately 4000 operations/sec with an average latency of 8 milliseconds (and 32 YCSB client threads). Further increasing the request rate does not increase the throughput, while the latency keeps increasing. This depicts an overloaded system, where the current number of servers cannot serve all incoming requests, leading to queueing effects. This behavior is same for both workloads.

6.2.2 Scalability

In our next experiments, we evaluated the scalability of CATS. We increased the dataset size and requests rate proportionally to the number of servers, as we did in the performance experiments. Figure 6.5 shows the throughput of the system as we vary the number of servers for both workloads. The figure shows that CATS scales linearly with a slope of one. With small number of servers, it is more likely that requests already arrive at one of the replicas for the requested key. Thus, the number of messages sent over the network is smaller. This explains the slightly higher throughput for 3 and 6 servers. The reason for linear scaling is that CATS is completely decentralized and all nodes are symmetric. Owing to linear scalability, the number of servers needed to achieve a certain throughput or to handle a certain rate of requests, can be calculated easily when deploying CATS in a cloud environment, provided the load is balanced across the servers. Such a decision can be made actively by either an administrator, or a feedback control loop that monitors the rate of client requests [110].

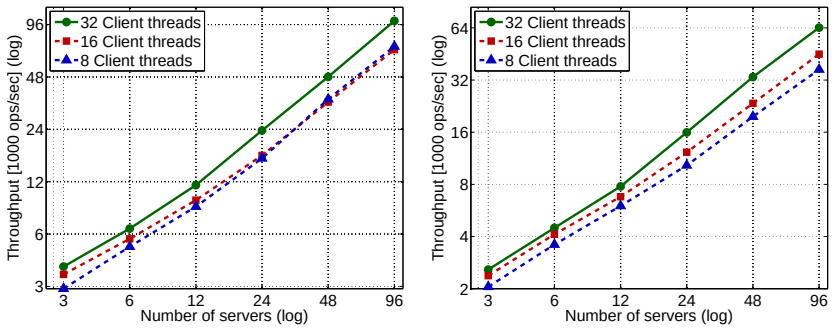


Figure 6.5: Scalability for a read-intensive (95%, left) and an update-intensive (50%, right) workload.

6.2.3 Elasticity

A highly desirable property for systems running in cloud environments is *elasticity*, the ability to add or remove servers while the system is running. When a system is overloaded, i.e. latency per operation is so high that it violates a service-level agreement (SLA), the performance can be improved by adding new servers. Similarly, when the load is very low, one can reduce running costs by decreasing the number of servers without violating any SLA. A system with good elasticity should perform better as servers are added, with a short disruption while the system reconfigures to include the new servers. The length of the disruption depends on the amount of data that needs to be transfered for the reconfiguration. A well-behaved system should have low latencies during this disruption window so that the end clients are not affected. In this experiment, we evaluated the elasticity of CATS. We started the system with 3 servers, loaded 2.4 million 1 KB values, and injected a high operation request rate via the YCSB client. While the workload was running and keeping the request rate constant, we added a new server every 10 minutes until the server count doubled to 6 servers. Afterwards, we started to remove servers every 10 minutes until we were back to 3 servers. We measured the average of operation latencies in 1 minute intervals throughout the experiment. The results of our experiment are presented in Figure 6.6. These results show that CATS incorporates changes in the number of servers with short windows (1–2 minutes) of disruption when the reconfiguration occurs, while the average latencies remain bounded by $2 \times x$ where x is the latency before the reconfiguration was triggered. Furthermore, since CATS is scalable, the latency approximately halves when the number of servers doubles to 6

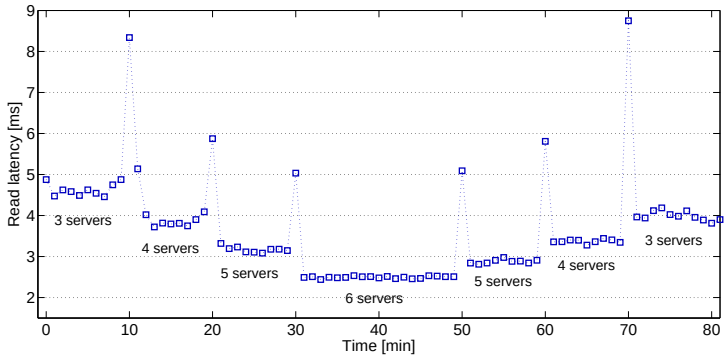


Figure 6.6: Elasticity for a read-only workload.

between 30–50 minutes compared to 3 servers between 0–10 minutes. As nodes are removed after 50 minutes, the latency starts increasing as expected.

6.2.4 Overhead of Atomic Consistency and Consistent Quorums

Next, we evaluated the overhead of atomic consistency compared to eventual consistency. For a fair comparison, we implemented eventual consistency in CATS, enabled through a configuration parameter. Here, read and write operations are always performed in one phase, and read-impose (read-repair in Cassandra terms) is never used. When a node n performs a read operation, it sends read requests to all replicas. Each replica replies with a timestamp and value. After n receives replies from a majority of replicas, it returns the value with the highest timestamp as a result of the read operation. Similarly, when a node m performs a write operation, it sends write requests to all replicas, using the current wall clock time as a timestamp. Upon receiving such a request, a replica stores the value and timestamp only if the received timestamp is higher than the replica's local timestamp. The replica then sends an acknowledgment to the writer m. Node m considers the write operation complete upon receiving acknowledgments from a majority of the replicas.

We also measured the overhead of consistent quorums. For these measurements, we modified CATS such that nodes did not send the replication group view in the read and write messages. Removing the replication group from messages reduces the size of messages, and hence

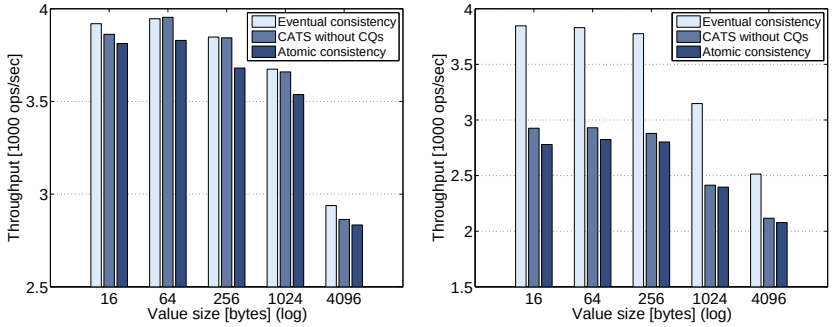


Figure 6.7: Overhead of Atomic consistency and Consistent quorums for a read-intensive (95%, left) and an update-intensive (50%, right) workload.

requires less bandwidth.

For these experiments, we varied the size of the stored values, and we measured the throughput of a system with 3 servers. Our results, averaged over five runs, are shown in Figure 6.7. The results show that as the value size increases, the throughput drops. This implies that the network becomes a bottleneck for larger value sizes. The same trend is observable in both workloads. Furthermore, as the value size increases, the cost of using consistent quorums becomes negligible. For instance, the loss in throughput for both workloads when using consistent quorums is less than 5% for 256 bytes values, 4% for 1KB values, and 1% for 4KB values.

Figure 6.7 also shows the cost of achieving atomic consistency by comparing the throughput of regular CATS with the throughput of our eventual consistency implementation. The results show that the overhead of atomic consistency is negligible for a read-intensive workload but as high as 25% for an update-intensive workload. The reason for this difference in behavior between the two workloads is that for a read-intensive workload, the second phase for reads (read-impose/read-repair) is rarely needed, since the number of concurrent writes to the same key are very low due to the large number of keys in the workload. For an update-intensive workload, due to many concurrent writes, the read operations often require to impose the read value. Hence, in comparison to an update-intensive workload, the overhead of achieving linearizability is very low (less than 5% loss in throughput for all value sizes) for a read-intensive workload. We believe that this is an important result. Applications that are read-intensive can opt for atomic consistency without a significant loss in performance, while avoiding the

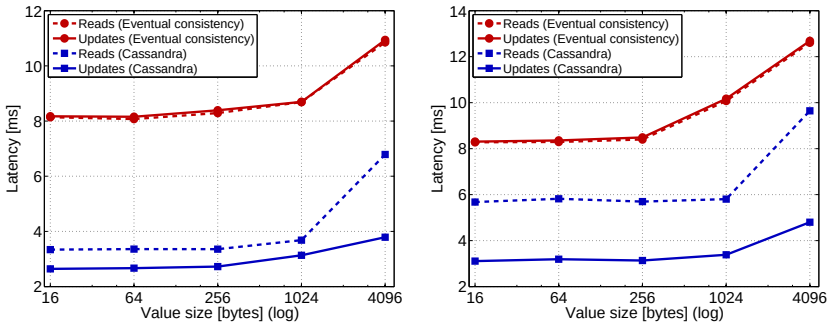


Figure 6.8: Comparison of latencies for Cassandra and CATS with Eventual Consistency for a read-intensive (95%, left) and an update-intensive (50%, right) workload.

complexities of using eventual consistency.

6.2.5 Comparison with Cassandra

The architecture of Cassandra [81] and Dynamo [38] are closest to CATS as both use consistent hashing with successor-list replication. Since Cassandra is available openly, here we compare the performance of CATS with that of Cassandra. We are comparing our research system with a system that leverages half a decade of implementation optimizations and fine tuning. Our aim is to give readers an idea about the relative performance difference, which, taken together with our evaluation of the cost of consistency, may give an insight into the cost of atomic consistency if implemented in Cassandra. We leave the actual implementation of consistent quorums in Cassandra to future work.

We used Cassandra 1.1.0 for our experiments, and used the `QUORUM` consistency level for a fair comparison with CATS. We chose the initial data size such that the working set would fit in main memory. Furthermore, since CATS only stores data in main memory while Cassandra uses disk, we set `commitlog_sync: periodic` in Cassandra for a fair comparison to minimize the effects to disk activity on operation latencies. Figure 6.8 shows a comparison of average latencies, averaged over five runs, for the same workloads for Cassandra and Eventual consistency implemented in CATS. The trend of higher latencies for large value sizes remains the same for both systems and workloads as the network starts to become a bottleneck. For CATS, read and write latencies are the same since both involve the same message complexity (one phase) and the same message sizes. On the other hand, Cassandra writes are

faster than reads, which is a known fact since writes require no reads or seeks, while reads may need to read multiple SSTables[6]. The results show that the operation latencies in CATS are approximately three times higher than in Cassandra (except reads in an update-intensive workload, where the effects of commit log disk accesses affect Cassandra's performance).

Given our comparison between Cassandra and Eventual consistency in CATS, and the low loss in throughput for achieving atomic consistency compared to eventual consistency (Section 6.2.4), we believe that an implementation of consistent quorums in Cassandra can provide linearizable consistency without considerable loss in performance (e.g. less than 5% loss for a read-intensive workload).

6.3 Discussion

Given the restrictions outlined by Brewer's Conjecture, different systems optimize for different aspects. At one extreme, there are systems like Cassandra, and Dynamo, that only guarantee eventual consistency but are always available (unless the whole system fails). On the other extreme, there are systems like CATS, Bigtable [25], Spinnaker [123], Spanner [31], and Scatter [53], that guarantee strong consistency. In the middle of this spectrum, several systems have been proposed that provide weaker consistency guarantees than strong consistency, yet still stronger than eventual consistency. Such systems aim to overcome the performance degradation (e.g., high latency) that comes with guaranteeing strong consistency [1]. These systems include PNUTS [29] (guarantees time-line consistency), and COPS [97] (guarantees causal consistency). Choosing which system to use depends on the application requirements. We aim to support applications that require strong consistency (yet scale linearly).

While systems such as Bigtable [25], and HBase [9], support consistency, they rely on a central server for coordination and data partitioning. Similarly, Spinnaker [123] uses Zookeeper [66] for coordination and data partitioning. Since these systems are centralized, their scalability is limited. In contrast, CATS is decentralized and all nodes are symmetric, allowing for unlimited scalability.

Similar to CATS, Scatter [53] is a scalable and consistent distributed key-value store. Scatter requires a distributed transaction across three adjacent replication groups for reconfiguration operations to succeed. In contrast, CATS has a simpler and more efficient, both in number of

[6] http://wiki.apache.org/cassandra/ArchitectureOverview

messages and message delays, reconfiguration protocol that does not need a distributed transaction. In CATS, each reconfiguration operation only operates on the replication group that is being reconfigured. Therefore, the period of unavailability to serve operations is much shorter (almost non-existent) in CATS, compared to Scatter. Furthermore, we focus on consistent-hashing at the node level, which makes our approach directly implementable in existing key-value stores like Cassandra [81]. Lastly, since we build CATS on top of ReCircle, our solution is able to handle network partitions and mergers.

Determining the set of replicas for a given object, or set of objects, in a dynamic setting is known as maintaining the *group membership*. Consensus is generally used to maintain, and make changes, to the replication set [99, 24]. Similarly, to handle dynamic environments, atomic registers were extended by protocols such as RDS [26] and DynaStore [6] to be reconfigurable. In our work, we employ consensus to reconfigure the replication groups. Compared to the afore-mentioned systems, which are not scalable as they do not partition the data across machines, we explore the complexities of, and propose a solution for, a scalable self-managing system.

In our work, we have shown that it is non-trivial to achieve linearizable consistency in dynamic, scalable, and self-organizing key-value stores which distribute and replicate data according to the principle of consistent hashing. We introduce consistent quorums as a solution to this problem for partially synchronous network environments prone to message loss, network partitioning, and inaccurate failure suspicions. We have built CATS, a distributed key-value store that leverages consistent quorums to provide linearizable consistency and partition tolerance. Our evaluation shows that it is feasible to provide linearizable consistency for those applications that do indeed need it, e.g., with less than 5% throughput overhead for read-intensive workloads.

CHAPTER 7

Replication

This chapter deals with the data level in overlays, i.e., how to manage data items stored in the overlay amid node joins and failures. We present a replication scheme, called *ID-Replication*. ID-Replication allows varied replication degrees in the system, and requests do not need to go through a master replica. It gives more control to an administrator and to implement policies, without hampering self-management. Furthermore, ID-Replication is less sensitive to churn compared to existing schemes, thus being well-suited for building consistent services in asynchronous networks where false failure detections are a norm. Since we use a generic design, ID-Replication can be used in any structured overlay network.

Introduction Structured overlay networks provide the infrastructure used to build scalable and fault-tolerant key-value stores, e.g. Cassandra [80], Dynamo [38], and Riak [13]. While scalability and self-management derives from consistent hashing [76], data fault-tolerance is achieved by replication. There are different strategies for replication in overlays, such as successor-list replication [153], using multiple hash functions [124, 164], and symmetric replication [48]. Out of these, successor-list replication is the most popular and widely used in ring-based overlays. For instance, overlays including Chord [153], Pastry [127] (with a minor modification), Accordion [92], Dynamo [38], and Cassandra [80], all use successor-list replication.

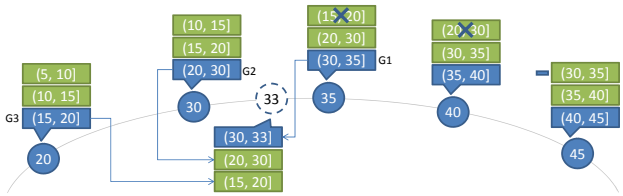


Figure 7.1: A new node 33 joins in a system using successor-list replication and degree 3. 6 nodes are involved in making changes, and 4 replication groups have to be updated.

7.1 Downsides of Existing Schemes

We introduced successor-list (SL) replication and symmetric replication in Section 2.2.1. In this section, we discuss short-comings of these schemes, thus motivating the need for another replication scheme.

Replica groups affected by churn: Churn - node joins and failures - is considered a norm in P2P systems. A desirable behaviour is that a churn event should not affect the configuration of an overlay greatly. In SL-replication, the unit of replication is a node's assigned key space, also known as the node's responsibility. For instance in Figure 2.2, the key space assigned to node 30 is $(20, 30]$, which is replicated on 35 and 40. Consequently, for a replication degree of r, each node replicates r node responsibilities going anti-clockwise.

When a new node joins the overlay, it divides a node responsibility range into two ranges. Similarly, a node failure results in merger of two node responsibilities. Since each node responsibility range is replicated on r nodes, and each node replicates r ranges, a single churn event results in reconfiguration of r replication groups. Furthermore, a join event involves action on behalf of $2 \times r$ nodes, and a failure involves action on behalf of $(2 \times r) - 1$ nodes. This is shown in Figure 7.1 where a new node 33 joins the system in a state shown in Figure 2.2. Replication groups $G1$, $G2$, and $G3$ need to be updated, and nodes 20, 30, 33, 35, 40, 45 are involved in such updates.

This approach has multiple drawbacks. First, a single churn event is overly complicated, involving many nodes. Second, consistent services built on top of overlays require consistent views of replication groups. For instance, Scatter [52], Etna [111], and CATS [22] require consensus whenever a replication group changes. A high number of reconfigurations for a single churn event is undesirable. We came across such

complexities with SL-replication while implementing CATS 6. Second, since a node replicates multi node responsibilities, it becomes difficult to implement different replication policies, e.g. replica placement, for different key ranges.

Load-balancing: We argue that SL-replication is complicated to load-balance. Consider an unbalanced system, such as the one depicted in Figure 2.2. It is unbalanced in-terms of keys since node 30 is responsible for storing 10 keys while all other nodes are responsible for storing 5 keys. A simple load-balancing mechanism, such as [75], would move node 30 counter-clockwise to handover responsibility of some keys to 35, or move 20 clockwise so that 20 takes over responsibility of some keys from 30. Since $keys \in (20, 30]$ are replicated on 3 nodes, such a movement will reduce load from one replica node only. Hence, r node movements on the identifier ring are needed to balance the load of one key range.

Security: In SL-replication, all requests for a key k end up on the node p responsible for k. This has two drawbacks. First, it is difficult to load-balance requests since all requests for k pass through p before they can be routed to a replica. Hence, p becomes a bottleneck. Second, if p is an adversary, it can launch a malicious attack against requests for k [146].

Bulk operations: Symmetric replication [48] is an alternate replication scheme for overlays. In Symmetric replication, the identifier space is partitioned into $\frac{N}{r}$ equivalence classes, and keys are stored symmetrically using equivalence classes. However, such a placement requires a complicated bulk operation [47] for retrieving replicas of all keys in a given range. Node failure handling has to use such a bulk operation to find data to be replicated. Such a bulk operation is complex, requires extra messages, and induces a delay before the churn event is completely catered.

Replication degree: SL-replication and Symmetric replication assume that the replication degree is constant throughout the system. For instance, the equivalence classes in Symmetric replication are created based on the system-wide replication degree r. This restricts popular/hot data or critical data from having more replicas than unpopular or less critical data.

We faced most of these downsides while using SL-replication to build a scalable key-value store in Chapter 6. For instance, for each

churn event or false failure suspicion by the failure detector, the number of reconfigurations were high, making it difficult to debug and analyze the system. This made it complicated to keep track of things happening in the system. Similarly, while using our key value store for an application, it was difficult to load-balance and configure replication policies, such as locality of replicas. This led to the design of ID-Replication.

7.2 ID-Replication

In this section, we describe a replication scheme for ring-based overlays, called ID-Replication. We first provide an overview of ID-Replication, give a detailed algorithmic specification, and then discuss its desirable properties.

7.2.1 Overview

We set out to design a replication scheme that is less sensitive to churn in terms of the number of replication groups that need to be reconfigured. In ID-Replication, we use sets of nodes, called *groups*, instead of individual nodes, as the building block for the overlay. Instead of partitioning the identifier space amongst nodes, we partition the identifier space among groups. Thus, compared to the simple structured overlay model where nodes are responsible for key ranges (Section 2.1.1), we assign responsibility ranges to groups. Consequently, groups are assigned identifiers from the identifier space. The idea of using groups instead of nodes as the building block of an overlay can be applied to the majority of the overlays. For the sake of simplicity, we use the ring-based Chord-like notation in this chapter.

All nodes within a group have the same identifier as the group. To distinguish nodes within a group, each node also has a group-local identifier. The group-local identifiers of nodes only need to be unique within the group. For efficient routing, each node maintains long range links, such as fingers in Chord.

The model of ID-Replication is shown in Figure 7.2. There are five groups on the identifier space: $20, 30, 35, 40$ and 45. The successor of a group is the first group encountered going clockwise from that group, e.g. group 40 is the successor of group 35. Similarly, the predecessor of a group is the first group encountered going anti-clockwise, e.g. 30 is the predecessor of 35. A group is responsible for the key range from its predecessor to itself, e.g. group 35 is responsible for $keys \in (30, 35]$.

Each group is composed of a number of nodes, e.g. group 30 contains nodes $\{1, 2, 3\}$. The nodes of a group are the replicas for the keys

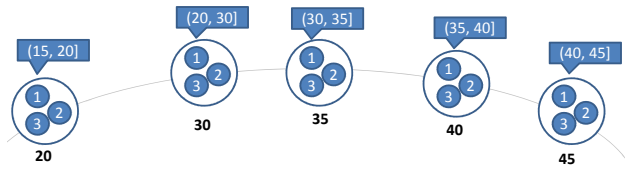


Figure 7.2: A configuration of ID-Replication. Replica groups are denoted by a single identifier on the identifier space ring. Nodes in a replica group $G1$ are responsible for storing keys between $G1$'s predecessor replica group's identifier and $G1$. Nodes within a replica group are differentiated by using group-local identifiers (1, 2, and 3 in the figure).

that the group is responsible for. The size of each group is specified using two parameters: r_{min} and r_{max}. Thus, the replication degree of keys is always between r_{min} and r_{max}. Instead of having a lower and upper bound, we discuss how to achieve a particular replication degree for a group in Section 7.5.

To maintain the ring under dynamism, we employ a modified version of periodic stabilization [153] that operates on groups instead of nodes. Furthermore, we use *gossiping* between nodes in a group to synchronize the view of the group among the group members.

We use two operations for reconfiguring groups: *Merge* and *Split*. When the size of a group $G1$ drops below r_{min}, we need to *merge* $G1$'s members with another group $G2$ such that the size of the merged group should be less than r_{max}. The merged group $G = G1 \cup G2$ retains the identifier of $G2$.

When the size of a group G becomes larger than r_{max}, we need to *split* it into two groups, $G1$ and $G2$, such that the size of each split group is larger than or equal to r_{min}. The identifiers of $G1$ and $G2$ are calculated in a way to increase the load-balance in the system.

A failure of a node can trigger a merge. Similarly, a new node joins an existing group, which can result in a split.

7.2.2 Algorithm

We give a full specification of ID-Replication as Algorithm 15 and 16. Each node stores a group-local identifier l_{id}, a group identifier id, and a set of nodes in its group, $group$. The successor $succ$ of a node is a tuple containing the identifier of the successor group, denoted as $succ.id$, and

the set of nodes in the successor group, denoted as *succ.group*. Similarly, the predecessor is a tuple of an identifier and group.

A new node p joins the system by attempting to become a member of a group of size less than r_{max} to avoid a split operation. Ideally, p should join the lowest-sized group. Such a group can be found in a best-effort manner by a random walk, gossiping, or by maintaining directories that store such information (as in [54]). Directories in a overlay are well-known identifiers/keys under which meta-data can be stored. The node through which p joins a group is denoted as *seed* in the algorithm. If the *seed* does not allow p to join the group, then p retries by searching for a different *seed* belong to a different group. If p is unable to find a group with less than r_{max} members after a number of retries, then p can join any group causing it to split into two.

Nodes maintain successor-lists to preserve the ring-geometry amid churn. The difference between Chord [153] and ID-Replication successor-lists is that the lists are composed of successive groups instead of successive nodes. If all nodes in the successor group of G fail or merge with another group, G points to the next group in the successor-list (Algo. 15, line 14). For ring and successor-lists maintenance, we use an algorithm similar to Chord's periodic stabilization (see Section 2.1.1), where nodes belonging to a group periodically stabilize the ring with nodes in their successor group (Algo. 15, lines 17–34).

Consider a group G with size, $|G|$, larger than r_{min}. Here, even if $|G| - r_{min}$ nodes leave the group, it will neither violate the replication degree, nor require a merge operation. These nodes [1] are called *standby* nodes. The standby nodes can potentially leave their group and join another group, e.g., to become part of a group in which a node has failed. To be used for such purposes, standby nodes advertise themselves (Algo. 16, lines 25–26) by either gossiping, or periodically updating their address information into directories (as in [54]). When using directories, a standby node s can randomly choose a key k from the well-known set of directory keys and advertise itself by storing s under k. The node in the overlay responsible for key k stores all such addresses of standby nodes that attempted to store their information under k, garbage collecting stale entries.

Each node p periodically checks if the size of its group, $|G|$, is between r_{min} and r_{max}. If $|G|$ is smaller than r_{min}, then the replication degree has dropped below the specified value. To maintain a replication degree of at least r_{min}, either a node should join the group, or

[1] nodes that can leave the group without violating the replication degree or requiring a merge operation

Algorithm 15 ID-Replication(Part I): Joins and failures

```
1:  upon event ⟨ Join | seed ⟩ at n                          ▷ retried with new seed
2:      sendto seed : JoinRequest⟨⟩
3:  end event

4:  receipt of JoinRequest⟨⟩ from m at n
5:      if |group| < r_max then                              ▷ if n's replica group has space
6:          group := group ∪ {m}
7:          sendto m : Move⟨id, group, succ, pred⟩
8:      end if
9:  end event

10: upon event ⟨ NodeFailure | f ⟩ at n                      ▷ Node f failed
11:     group := group − {f}
12:     pred.group := pred.group − {f}
13:     succ.group := succ.group − {f}
14:     if succ.group = ∅ then
15:         succ := nextInSuccessorGroups()
16:     end if
17: end event

    ▷ Periodically check for new successor and predecessor groups
18: every δ time units at n
19:     r_s := selectRandom(succ.group)
20:     sp_x := r_s.GetPredecessor()                         ▷ fetch remotely
21:     if sp_x.id ∈ (id, succ.id) then
22:         ⟨succ.id, succ.group⟩ := ⟨sp_x.id, ps_x.group⟩
23:     end if
24:     r_u := selectRandom(succ.group)
25:     succ.group := r_u.GetGroup()                         ▷ fetch remotely
26:     for all p ∈ succ.group do
27:         sendto p : Notify⟨id, group⟩
28:     end for
29: end event

30: receipt of Notify⟨id_m, group_m⟩ from m at n
31:     if id_m ∈ (pred.id, id) or m ∈ pred.group then
32:         ⟨pred.id, pred.group⟩ := ⟨id_m, group_m⟩
33:     end if
34: end event
```

Algorithm 16 ID-Replication(Part 2): Split and Merge operations

▷ Periodically attempt to keep $r_{min} < |group| < r_{max}$
1: **upon** $|group| > r_{max}$ **at** n ▷ Split operation
2: $split := getTop(\text{sort}(group), r_{min})$ ▷ get r_{min} nodes with lowest l_{id}
3: $retain := group - split$
4: **for all** $p \in split$ **do**
5: **sendto** p : Move$\langle newGroupKey, split, retain, pred\rangle$
6: **end for**
7: **for all** $p \in retain$ **do**
8: **sendto** p : Move$\langle id, retain, succ, split\rangle$
9: **end for**
10: **end event**

11: **upon** $|group| < r_{min}$ **at** n ▷ due to failures
12: $node :=$ SearchStandbyNode()
13: **if** $node = nil$ **then** ▷ search failed, Merge with successor group
14: $id_{new} := succ.id$
15: $group_{new} := succ.group \cup group$
16: $succ_{new} := succ$.GetSuccesor() ▷ fetch remotely
17: **for all** $p \in group_{new}$ **do**
18: **sendto** p : Move$\langle id_{new}, group_{new}, succ_{new}, pred\rangle$
19: **end for**
20: **else** ▷ make the standby node part of n's group
21: **sendto** $node$: Move$\langle id, group \cup \{node\}, succ, pred\rangle$
22: **end if**
23: **end event**

24: **every** δ **time units at** n
25: **if** $indexOf(n, sort(group)) > r_{min}$ **then**
26: AvertiseAsStandbyNode(n)
27: **end if**
28: GossipView($group$) ▷ sync view (& data) with group members
29: **end event**

30: **receipt of** Move$\langle id_x, group_x, succ_x, pred_x\rangle$ **from** m **at** n
31: HandoverData()
32: $id := id_x$
33: $group := group_x$
34: $succ := succ_x$
35: $pred := pred_x$
36: **end event**

the group should merge with another group. p searches for a standby node by gossiping or contacting a directory, and tries to include the found standby node in its (p's) group. If a standby node cannot be found, p triggers a merge operation for its group to become part of another group (Algo. 16, lines 11–23). Similarly, if $|G|$ is larger than r_{max}, p initiates a split operation by dividing the group into two groups (Algo. 16, lines 1–10). Different policies can be employed for determining the identifier of the newly created group. One such policy is use an identifier which will split the load equally amongst the two groups.

Due to joins, failures, false failure detection, and standby nodes moving to new groups, the view of a group may diverge at the nodes in a group. To over come such differences, each node p periodically gossips with its group members to synchronize their view of the group [162]. Furthermore, the gossiping can potentially use anti-entropy [119, 157] to update data items within the group.

7.3 Evaluation

In this section, we evaluate ID-Replication, and compare it to successor-list replication using stochastic discrete-event simulation in Kompics [10]. We use the King dataset [59] for message latencies between nodes. Each experiment has the following structure: we initialize the overlay with 2000 nodes. Once the overlay converges, we subject it to 2000 churn events (1000 joins and 1000 failures), and measure various metrics till the topology converges. The rate of churn is governed by the lifetimes of the nodes. In our experiments, the lifetimes of the nodes had a poisson distribution, and each node failure was followed by a join event to maintain a network size of 2000. We evaluate both replication schemes under various levels of churn by changing the median of poisson distribution for the lifetimes. Here, a higher median lifetime results in a lower churn rate. We performed simulations for periodic stabilization periods of 30 and 60 seconds, but only present graphs for 60 seconds since the experiment results for both stabilization rates were the same We simulate 3 directories for nodes to publish/advertise and find standby nodes, and use a value of $r_{max} = (2 \times r_{min}) - 1$. We repeated each experiment for 10 different seeds and report the averages.

7.3.1 Replication groups restructured

To evaluate the effectiveness of ID-replication, we measure and compare the number of replication groups that need to be reconfigured due to churn events. The results are shown in Figure 7.3. The x-axis

shows the median lifetime used for nodes, and the y-axis depicts the number of replication groups restructured per churn event ($\mathcal{R}_c$). As analyzed earlier, the figure shows that for SL-replication, $r = \mathcal{R}_c$, while for ID-Replication, the value of $\mathcal{R}_c$ is close to one. Hence, $\mathcal{R}_c$ is dependent on the replication degree in SL-replication. On the other hand, ID-Replication does not depend on r because a churn event effects one or two groups regardless of the replication degree. Only one replication group restructuring is needed when: (1) a new node p joins a group G such that $|G \cup \{p\}| \leq r_{max}$, i.e., the join does not require splitting the group, or (2) a node $p \in G$ fails such that $|G - \{p\}| \geq r_{min}$, i.e., the failure does not require merging the group. Two replication groups have to restructured when a Split or Merge is required, or when a standby node has to be moved from one group to another.

Next, we measure the number of Split, Merge, and standby node movements required per churn event. While this is interesting to evaluate since these operations result in the reconfiguration of two replica groups, they also affect the amount of data that needs to be transferred. A Split operation does not require any data movement as all the nodes in the newly created groups already have the data for both groups. Here, the data responsibility of each node is merely reduced after the split. A Merge operation is costly in terms of data transfer as it results in data exchange of two responsibility ranges by all members of the two groups being merged[2]. On the other hand, the movement of a standby node requires data transfer of one responsibility range only once to the node being moved. The number of splits, merges, and standby node move-

[2]In a well-managed system, the number of merge operations can be reduced by having multiple routing virtual servers (see Section 7.5) in each group.

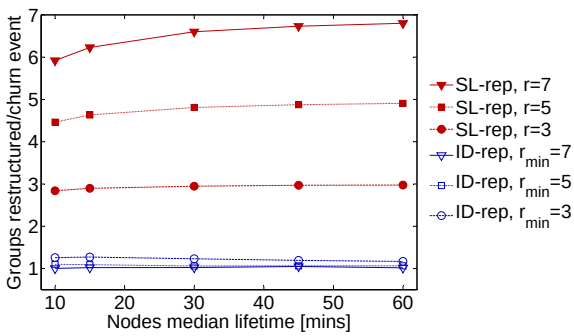


Figure 7.3: Number of replication groups restructured per churn event.

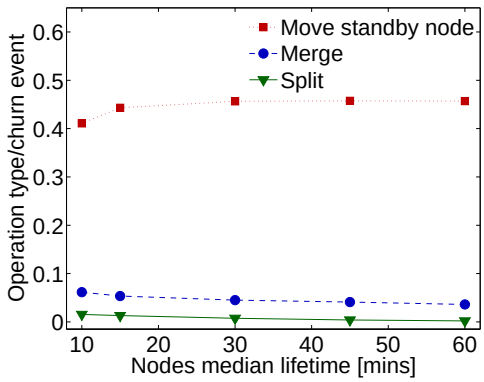


Figure 7.4: The number of standby node movements increases with decreasing churn rate, thus reducing group merges.

ments per churn event is shown in Figure 7.4, which illustrates that the ratio of Split and Merge operations is low.

7.3.2 Nodes involved in updates

Each churn event requires action on behalf of a certain number of nodes. In this experiment, we count the number of nodes involved in reconfiguration operations. This count is shown in Figure 7.5, normalized against the number of churn events ($\mathcal{R}_n$). As analyzed in Section 7.1, for SL-replication, $\mathcal{R}_c$ approaches $2 \times r$. Since the number of groups reconfigured in ID-Replication is close to one for a churn event, $\mathcal{R}_n$ is close to r. It is noteworthy that the value of $\mathcal{R}_n$ for ID-Replication drops as the mean life time of nodes increases, which is the opposite for SL-replication. At lower churn rates when using ID-Replication, the number of splits and merges is reduced because each group can have standby nodes, and new nodes can take better decisions about which group to join. This reduces splits and merges, thus resulting in fewer nodes involved at low churn rates. Such a characteristic makes ID-Replication well-suited for managed environments, such as data-centers, where the rate of churn is low.

7.4 Related work

Scatter [52] uses a similar scheme to ID-Replication for achieving consistency in DHTs. Compared to our scheme, they further sub-divide the groups to differentiate between key responsibilities of each node.

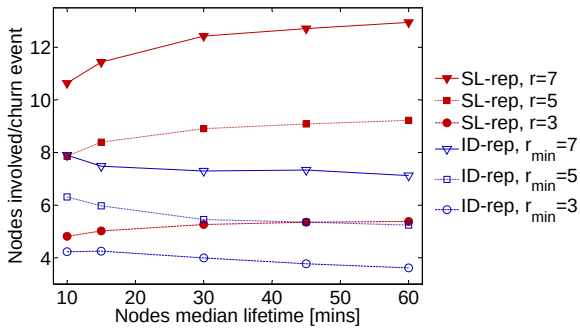


Figure 7.5: Number of nodes involved in updates for each churn event.

Furthermore, they do not evaluate or argue for the usefulness of their scheme. We provide algorithmic specification of our work, backed by design decisions and evaluation with comparison to SL-replication.

Agyaat [145] proposes to use groups of nodes, called *clouds*, to provide mutual anonymity in structured overlays. Compared to ID-replication, Agyaat maintains an R-Ring and an overlay with the clouds, which is more complicated and requires some nodes to be part of two overlays. A similar approach is taken by Narendula et al. [114], where nodes form sub-overlays with trusted nodes for better access control in P2P data management.

7.5 Discussion

As we discuss in Section 7.2.1 and evaluate in Section 7.3, ID-Replication requires less replication group reconfigurations per churn event compared to SL-replication. This makes ID-Replication ideal for building a consistent key-value store. Each replication group can be considered as a replicated state machine [132] and operations are performed on the data in a total order within the group. To handle dynamism, we need to support the merge and split operations where the view of a group changes. For this, we can use a reconfigurable replicated state machine [88], such as SMART [98]. Using SMART with SL-replication is both complicated and expensive as replicated state machines are implemented using consensus [86]. Since ID-Replication requires fewer replication group reconfigurations per churn event, it will require fewer instances of consensus. The amount of data that needs to transfered per churn event, however, remains the same, though the coordination becomes simpler and cheaper. Apart from joins and failures, false fail-

ure suspicions are common in asynchronous networks, which will trigger much more unnecessary reconfiguration requests in SL-replication than in ID-Replication[3].

ID-Replication allows the system user to have different replication degrees for different key ranges. We use two parameters, r_{min} and r_{max}, to control the replication degree, which can be different for different replication groups. For a given key range, the number of replicas is at least r_{min} and at most r_{max}. Thus, popular or critical data can have more copies than other data by setting higher values of r_{min} and r_{max} for the corresponding key range. We propose using virtual servers [33] for maintaining a certain replication degree r instead of a varying replication degree between r_{min} and r_{max}. Assuming $r_{min} \leq r \geq r_{max}$, the first r nodes with the lowest local identifiers in a group can be used to replicate the data items. The rest of the $r_{max} - r$ nodes, if any, can still participate at the routing level (performing gossiping, periodic stabilization, forwarding lookups) but instead of storing data, they can act as standby nodes for any failures of replicas storing the data. A node can host some virtual servers that store data as well as participate in routing, and some virtual servers that only participate at the routing level.

Owing to the design of ID-Replication, an administrator has much more control over the system compared to SL-replication, which simplifies implementing various policies for replication. For instance, the administrator can control how many and which machines should serve a particular key-range. This allows the usage of specialized hardware for handling requests for certain keys, and configuring locality for replicas. On the contrary, a node in SL-replication is responsible for replicating multiple key ranges (r key partitions anti-clockwise), making it harder to control which nodes replicate some keys as consecutive key ranges on the ring may have different requirements. For example, consider Figures 2.2 and 7.2. Assume we want to store keys $k \in (21, 30]$ on machines of type (or location) $\mathcal{A}$, and keys $j \in (31, 35]$ on machines of type (or location) $\mathcal{B}$. This is difficult to achieve in SL-replication as the node responsible for j is also a replica for k. On the other hand, achieving the afore-mentioned is much simpler in ID-Replication as $\mathcal{A}$ machines can join the group 30 and $\mathcal{B}$ machines can join group 35.

Routing tables, e.g. fingers in Chord, can also be build using groups. Each routing pointer can point to a group, containing addresses of multiple nodes. Greedy routing can be done on group identifiers, and a

[3]We came across such issues while building a consistent key-value store on top of an overlay using successor-list replication (Chapter 6)

lookup can be routed to a random node in the group. For fault-tolerance and better performance, a lookup can be routed by forwarding in parallel to all nodes in the groups at each hop, and considering only the first reply. While such a mechanism consumes more bandwidth, it (a) is more reliable as it can tolerate failure of nodes in the path, and (b) has lower latency as the lookup can exploit multiple paths. Such parallel lookup techniques have also been proposed for Chord like overlays [90].

ID-Replication maintains groups using a gossip protocol, similar to Cyclon [160], which adds to the maintenance cost compared to only using periodic stabilization. Cyclon is inexpensive, especially given that the group sizes are small in our case and the churn rates are moderate in cloud environments. If the gossip rate is equal to the periodic stabilization rate, then the maintenance cost for ID-Replication is twice[4] than a system using only periodic stabilization. We also confirmed this via simulations. In data center environments, the maintenance cost can be reduced by choosing a low rate of gossiping since the churn rate is low.

[4] the cost is still moderate and negligible given today's interconnects

CHAPTER 8

Conclusion

Structured overlay networks are widely used as the building block for large-scale applications to locate resources. Any long-lived Internet-scale distributed system is bound to face network partitions. Hence, apart from dealing with normal churn rates, we argue that structured overlays should intrinsically be able to handle rare but extreme events such as network partitions and mergers, bootstraping, and flash crowds.

In this thesis, we have proposed techniques for identifying when a network partition heals, and have presented a simple and a gossip-based algorithm, called *Ring-Unification*, for merging similar ring-based structured overlay networks after the underlying network merges. Ring-Unification is an add-on algorithm, that works in conjunction with an overlay maintenance algorithm. To avoid the complexity of having multiple algorithms for catering various cases, we have presented *ReCircle*, an overlay algorithm that acts as an overlay maintenance algorithm, as well as deals with all extreme churn events such as bootstrapping, and network partitions and mergers. Under normal execution, ReCircle exchanges messages periodically like any other overlay maintenance protocol. On the other hand, we designed ReCircle to be reactive to extreme events so that it can converge faster when such events occur. Both, Ring-Unification and ReCircle, provides tunable knobs to trade-off between cost (bandwidth consumption) and performance (time to convergence) while handling extreme scenarios.

The CAP Theorem [19, 51] states that in a distributed asynchronous network, it is impossible to achieve consistency, availability and partition tolerance at the same time. Hence, system designers have to chose

two out of the three properties. Since network partitions are unavoidable in a distributed environment, we believe Ring-Unification and ReCircle are pivotal for structured overlay networks. In-line with the CAP Theorem, the choice between consistency and availability depends on the target application. For some applications, availability is of utmost importance, while weaker consistency guarantees (e.g. eventual consistency [38]) suffice. For others, data consistency is a must, even if the application is temporarily unavailable.

We have shown that in structured overlay networks, data inconsistency arises from lookup inconsistencies. We have studied the frequency of lookup inconsistencies, and found that its main cause is inaccurate failure detectors. Hence, the choice of a failure detection algorithm is of crucial importance in overlays. We have discussed and evaluated two techniques to reduce the affects of lookup inconsistencies: *local responsibilities* and using *quorum-based* algorithms. While the effects of lookup inconsistencies can be reduced by using local responsibilities, we show that using responsibility of keys may affect availability of keys and any corresponding data stored against the keys. This is a trade-off between availability and consistency. Next, we have shown that using quorum-based techniques amongst replicas of data items further reduces lookup inconsistencies. Since majority-based quorum techniques require a majority of the replicas to make progress, these algorithms may still make progress even with unavailability of some nodes/replicas. An application can employ both, local responsibilities and quorum-based algorithms, for reducing inconsistencies and achieving high availability.

There is a class of applications that require strong consistency guarantees and partition tolerance, even if the system becomes unavailable under certain failure scenarios. In this thesis, we have shown that it is non-trivial to achieve strong/linearizable consistency in dynamic, scalable, and self-organizing storage systems built on top of ring-based structured overlays, i.e., using the principle of consistent hashing [76]. We have built a key-value store, called *CATS*, employing principles from structured overlay networks, that provides strong consistency and partition tolerance. On the routing level of CATS, we use ReCircle for overlay maintenance amid churn, network partitions and mergers, and *local responsibilities* and *quorum-based* algorithms for reducing inconsistencies and achieving high availability. On the data level, in CATS, we have introduced *consistent quorums* for achieving linearizable/strong consistency in partially synchronous network environments prone to message loss, network partitioning, and inaccurate failure suspicions. For this

purpose, our solution employs consistent quorums along with various distributed algorithms, such as Consensus [86], ABD registers [11], and dynamic reconfiguration [88, 5]. We have described the design, implementation, and evaluation of CATS, a distributed key-value store that leverages consistent quorums to provide linearizable consistency and partition tolerance. CATS is self-managing, elastic, and it exhibits unlimited linear scalability, all of which are key properties for modern cloud computing storage middleware. Our evaluation shows that it is feasible to provide linearizable consistency for those applications that do indeed need it, e.g., with less than 5% throughput overhead for read-intensive workloads.

While building CATS, a strongly consistent key-value store, we came across various drawbacks of existing replication schemes used in structured overlays. In this thesis, we have discussed popular approaches employed for replication in structured overlays, including successor-list replication and symmetric replication, and outlined their drawbacks. Such drawbacks include the huge effect of churn on replication groups, dynamic load-balancing, and in-ability to have different replication degree for different key ranges (data items). We have presented the design, algorithmic specification, and evaluation of *ID-Replication*, a replication scheme for structured overlays that does not suffer from the aforementioned problems. ID-Replication does not require requests to go through a particular replica. Furthermore, it allows different replication degrees for different key ranges. This allows for using higher number of replicas for hotspots and critical data. Our results show that ID-Replication is less sensitive to churn than SL-replication, which makes it better suited for building consistent services, and for usage in asynchronous networks where inaccurate failure detections are a norm.

Knowledge of the current network size of a structured peer-to-peer system is a prime requirement for many systems, which prompted to finding solutions for size estimation. Previous studies have shown that gossip-based aggregation algorithms, though being expensive, produce accurate estimates of the network size. We have demonstrated the shortcomings in existing aggregation approaches for network size estimation and have presented a solution for ring-based structured overlays that overcomes the deficiencies. We have argued for an adaptive approach to convergence in gossip-based aggregation algorithms, instead of a predefined duration. Our solution is resilient to massive node failures and is aimed to work on non-random topologies, such as structured overlay networks.

8.1 Future work

We believe that it is interesting to investigate whether gossip-based topology generators, such as T-man [68] and T-chord [109], can be used to handle network mergers on the overlay level. These services, however, make use of an underlying membership service, such as Cyclon [160], Scamp [45], or Newscast [69]. Hence, one has to first investigate how well such membership services recover from network partitions (we believe this to be interesting in itself). Thereafter, one can explore how such topology generators can be incorporated into a structured overlay network.

Mathematical analysis of gossip-protocols is often done through simple recurrence relations or by using Markov chains, where the state of the chain can be the number of infected nodes [42]. The overlay merging algorithms we have proposed, Ring-Unification and ReCircle, mix deterministic overlay algorithms with that of gossip protocols. Consequently, we believe that an analysis of our algorithms will require modeling the routing pointers of every node as part of the chain state. We solicit such an analysis and believe it is an interesting future direction for this research.

During the merger of multiple overlays, some keys (and associated data items) may temporarily become unavailable due to inconsistent routing pointers. For instance, during merger, the finger pointers from a node in overlay $\mathcal{A}$ may point to a node in the other overlay $\mathcal{B}$. Here, a lookup request generated in $\mathcal{A}$ may end up on a node in $\mathcal{B}$, hence not finding the lookup key. While such inconsistencies are rare and short-lived, we believe an extended analysis of this issue to be a potential future direction. Such an analysis has been done by Datta [36], and can potentially be extended to our overlay merging algorithms.

In the context of key-value stores built on top of overlays, a merger of multiple overlays can result in a large amount of data transfer amongst nodes from different overlays. One can employ a lazy (background) data transfer mechanism to avoid congesting the network, while keeping forwarding pointers at each node to be able to access the data before it has been transferred to the newly responsible node. Such techniques, including optimizations, require further investigation and are left as future work.

We introduced consistent quorums to provide a consistent view of replication groups. In CATS, we used consistent quorums to build a key-value store. Instead of only single-item linearizable operations, such consistent views can be leveraged further to implement distributed multi-item transactions. This will make CATS attractive for usage by

many cloud applications that require transactions. Furthermore, CATS can be extended to be a feature-rich data storage system by supporting column-oriented APIs, indexing, and search. Lastly, as the underlying key-to-node responsibility mapping mechanism (consistent hashing) used by CATS is the same as Cassandra, we believe our ideas can easily be implemented in Cassandra. Such an extension of Cassandra can potentially be extremely valuable for the open source community, and everyone using Cassandra. The applications using Cassandra do not need to be changed, and can benefit from linearizability by not worrying about the complexities that come with eventual consistency, at a small performance cost.

Due to its low sensitivity to churn, a possible step forward would be to build a consistent key-value store using ID-Replication. Each replication group can act as a replicated state machine, where operations are performed in a total order on the replicas (one such approach was attempted in Scatter [53]). Since replica groups change with dynamism, we propose using a reconfigurable replicated state machine, for instance, SMART [98]. Such a key-value store will directly support transactions across all items stored within a group since it orders operations within a replication group.

The local state initialization mechanism used in our network size estimation algorithm works for ring-based structured overlay networks. An interesting future direction is to extend the algorithm for usage with other geometries, e.g., XOR in Kademlia [106], or torus in CAN [124]. Furthermore, our algorithm can be generalized to calculate aggregates other than network size estimation, e.g., average load in the system. In our future work, we aim at extending our size estimation results to non-uniform distributed identifiers and performing extended costs analysis of the algorithm.

Bibliography

[1] Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 45:37–42, 2012.

[2] K. Aberer, L. O. Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth. The Essence of P2P: A Reference Architecture for Overlay Networks. In *Proceedings of the 5th International Conference on Peer-To-Peer Computing (P2P'05)*, pages 11–20. IEEE Computer Society, August 2005.

[3] 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*, 32(3):29–33, 2003.

[4] Skype active users. http://blogs.skype.com/en/2012/03/35_million_people_concurrently.html, April 2013.

[5] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 102:84–108, 2010.

[6] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58:7:1–7:32, April 2011.

[7] L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In *Proceedings of the 3rd International Workshop on Global and Peer-To-Peer Computing on Large Scale Distributed Systems (CCGRID'03)*, pages 344–350, Tokyo, Japan, May 2003. IEEE Computer Society.

[8] L. O. Alima, A. Ghodsi, and S. Haridi. A Framework for Structured Peer-to-Peer Overlay Networks. In *Post-proceedings of Global Computing*, volume 3267 of *Lecture Notes in Computer Science (LNCS)*, pages 223–250. Springer Verlag, 2004.

[9] Apache HBase. http://hbase.apache.org/, 2012.

[10] Cosmin Arad, Jim Dowling, and Seif Haridi. Developing, simulating, and deploying peer-to-peer systems using the kompics component model. In *Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE*, COMSWARE '09, pages 16:1–16:9, New York, NY, USA, 2009. ACM.

[11] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[12] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *PVLDB*, 5(8):776–787, 2012.

[13] Basho Riak. http://wiki.basho.com/Riak.html/, 2012.

[14] BerkeleyDB. http://www.oracle.com/technology/products/berkeley-db/, 2012.

[15] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the ACM SIGCOMM 2004 Symposium on Communication, Architecture, and Protocols*, pages 353–366, Portland, OR, USA, March 2004. ACM Press.

[16] Egypt's big Internet disconnect 2011, April 2013. http://www.guardian.co.uk/commentisfree/2011/jan/31/egypt-internet-uncensored-cutoff-disconnect.

[17] Andreas Binzenhöfer, Dirk Staehle, , and Robert Henjes. On the fly estimation of the peer population in a chord-based p2p system. In *19th International Teletraffic Congress (ITC19)*, Beijing, China, Sep. 2005.

[18] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.

[19] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[20] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[21] F. Jahanian C. Labovitz, A. Ahuja. Experimental Study of Internet Stability and Wide-Area Backbone Failures. Technical Report CSE-TR-382-98, University of Michigan, November 1998.

[22] CATS. http://cats.sics.se, April 2013.

[23] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[24] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[26] Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial, and Alex A. Shvartsman. Reconfigurable distributed storage for dynamic networks. *J. Parallel Distrib. Comput.*, 69:100–116, January 2009.

[27] The Cogent-Level 3 Dispute 2005. http://www.isp-planet.com/business/2005/cogent_level_3.html, December 2011.

[28] B. Cohen. Incentives Build Robustness in BitTorrent. In *First Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[29] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[30] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[31] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012.

[32] F. Dabek. *A Distributed Hash Table*. PhD dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, September 2005.

[33] F. Dabek, M. F. Kaashoek, D. R. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 202–215, Chateau Lake Louise, Banff, Canada, October 2001. ACM Press.

[34] A. Datta. Merging Intra-Planetary Index Structures: Decentralized Bootstrapping of Overlays. In *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, pages 109–118, Boston, MA, USA, July 2007. IEEE Computer Society.

[35] A. Datta and K. Aberer. The Challenges of Merging Two Similar Structured Overlays: A Tale of Two Networks. In *Proceedings of the First International Workshop on Self-Organizing Systems (IWSOS'06)*, volume 4124 of *Lecture Notes in Computer Science (LNCS)*, pages 7–22. Springer-Verlag, 2006.

[36] Anwitaman Datta. Merging ring-structured overlay indices: toward network-data transparency. *Computing*, 94(8-10):783–809, 2012.

[37] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, 1985.

[38] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[39] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, New York, NY, USA, 1987. ACM Press.

[40] Distributed *k*-ary System. http://dks.sics.se, 2006.

[41] eMule. http://www.emule-project.net/, March 2009.

[42] P. Th. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[43] Alex Feinberg. Project Voldemort: Reliable distributed storage. In *ICDE 2011*, 2011. http://project-voldemort.com.

[44] Cisco Visual Networking Index: Forecast and 2011-2016 Methodology. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360.pdf, November 2012.

[45] A. J. Ganesh, A.-M. Kermarrec, and L Massoulié. SCAMP: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication. In *Proceedings of the 3rd International Workshop on Networked Group Communication (NGC'01)*, volume 2233 of *Lecture Notes in Computer Science (LNCS)*, pages 44–55, London, UK, 2001. Springer-Verlag.

[46] Sanjay Ghemawat and Jeff Dean. LevelDB. http://code.google.com/p/leveldb/, 2012.

[47] A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH—Royal Institute of Technology, Stockholm, Sweden, December 2006.

[48] A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems . In *Proceedings of the 3rd International VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'05)*, volume 4125 of *Lecture Notes in Computer Science (LNCS)*, pages 74–85. Springer-Verlag, 2005.

[49] A. Ghodsi and S. Haridi. Atomic Ring Maintenance for Distributed Hash Tables. Technical Report T2007:05, Swedish Institute of Computer Science (SICS), 2007.

[50] David K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM Press.

[51] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[52] L. Glendenning, I. Beschastnikh, and A. Krishnamurthy. Scalable Consistency in Scatter. In *ACM SOSP*, pages 15–28, 2011.

[53] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 15–28, New York, NY, USA, 2011. ACM.

[54] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured P2P systems. In *Proceedings of the 23rd Conference of the IEEE Computer and Communications Societies*, 2004.

[55] P. B. Godfrey and I. Stoica. Heterogeneity and Load Balance in Distributed Hash Tables. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'05)*, pages 596–606, Miami, FL, USA, March 2005. IEEE Computer Society.

[56] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[57] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, October 2003. ACM Press.

[58] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the ACM SIGCOMM 2003 Symposium on Communication, Architecture, and Protocols*, pages 381–394, New York, NY, USA, 2003. ACM Press.

[59] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: estimating latency between arbitrary internet end hosts. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 5–18, New York, NY, USA, 2002. ACM.

[60] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[61] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 160–169, Berkeley, CA, USA, 2003. Springer-Verlag.

[62] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, WA, USA, March 2003. USENIX.

[63] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[64] K. Horowitz and D. Malkhi. Estimating network size from local information. *Information Processing Letters*, 88(5):237–243, 2003.

[65] G. Huang. Experiences with PPLive. Key note talk at Peer-to-Peer Streaming and IP-TV Workshop Invited held with ACM Sigcomm, Kyoto, Japan, August 2007.

[66] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, Berkeley, CA, USA, 2010. USENIX Association.

[67] ISP Quarrel Partitions Internet 2008. http://www.wired.com/threatlevel/2008/03/isp-quarrel-par/, April 2013.

[68] M. Jelasity and Ö. Babaoglu. T-man: Gossip-based overlay topology management. In *Proceedings of 3rd Workshop on Engineering Self-Organising Systems (EOSA'05)*, volume 3910 of *Lecture Notes in Computer Science (LNCS)*, pages 1–15. Springer-Verlag, 2005.

[69] M. Jelasity, W. Kowalczyk, and M. van Steen. Newscast Computing. Technical Report IR–CS–006, Vrije Universiteit, November 2003.

[70] M. Jelasity and A. Montresor. Epidemic-Style Proactive Aggregation in Large Overlay Networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 102–109, Tokyo, Japan, March 2004. IEEE Computer Society.

[71] M. Jelasity, A. Montresor, and Ö. Babaoglu. Gossip-based Aggregation in Large Dynamic Networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3), August 2005.

[72] Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, 2009.

[73] M. F. Kaashoek and D. R. Karger. Koorde: A Simple Degree-optimal Distributed Hash Table. In *Proceedings of the 2nd Interational Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 98–107, Berkeley, CA, USA, 2003. Springer-Verlag.

[74] D. R. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC'97)*, pages 654–663, New York, NY, USA, May 1997. ACM Press.

[75] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *IPTPS*, pages 131–140, 2004.

[76] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[77] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *44th Symp. on Foundations of Computer Science (FOCS)*, 2003.

[78] D. Kostoulas, D. Psaltoulis, I. Gupta, K. Birman, and A. J. Demers. Decentralized schemes for size estimation in large and dynamic groups. In *4th IEEE International Symp. on Network Computing and Applications (NCA 2005)*, pages 41–48, 2005.

[79] Gerald Kunzmann and Andreas Binzenhöfer. Autonomically Improving the Security and Robustness of Structured P2P Overlays. In *Proceedings of the International Conference on Systems and Networks Communications (ICSNC 2006)*, Tahiti, French Polynesia, October-November 2006. IEEE Computer Society.

[80] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, apr 2010.

[81] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[82] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering (TSE)*, 3(2):125–143, 1977.

[83] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[84] L. Lamport. On interprocess communication, part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[85] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[86] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[87] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[88] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, March 2010.

[89] B. Leong and J. Li. Achieving One-Hop DHT Lookup and Strong Stabilization by Passing Tokens. In *12th International Conference on Networks (ICON'04)*, Singapore, November 2004. IEEE Computer Society.

[90] B. Leong, B. Liskov, and E. Demaine. EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management. In *12th International Conference on Networks (ICON'04)*, Singapore, November 2004. IEEE Computer Society.

[91] J. Li. *Routing Tradeoffs in Dynamic Peerŋtoŋpeer Networks*. PhD dissertation, MIT—Massachusetts Institute of Technology, Massachusetts, USA, November 2005.

[92] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, Boston, MA, USA, May 2005. USENIX.

[93] J. Li, J. Stribling, R. Morris, M. F. Kaashoek, and T. M. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'05)*, Miami, FL, USA, March 2005. IEEE Computer Society.

[94] X. Li, J. Misra, and C. G. Plaxton. Brief Announcement: Concurrent Maintenance of Rings. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC'04)*, page 376, New York, NY, USA, 2004. ACM Press.

[95] D. Liben-Nowell, H. Balakrishnan, and D. R. Karger. Analysis of the Evolution of Peer-to-Peer Systems. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC'02)*, pages 233–242, New York, NY, USA, 2002. ACM Press.

[96] D. Liben-Nowell, H. Balakrishnan, and D. R. Karger. Observations on the Dynamic Evolution of Peer-to-Peer Networks. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS'02)*, volume 2429 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2002.

[97] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[98] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The smart way to migrate replicated stateful services. In *EuroSys*, 2006.

[99] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The smart way to migrate replicated stateful services. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 103–115, New York, NY, USA, 2006. ACM.

[100] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[101] N. A. Lynch, D. Malkhi, and D. Ratajczak. Atomic Data Access in Distributed Hash Tables. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Lecture Notes in Computer Science (LNCS), pages 295–305, London, UK, 2002. Springer-Verlag.

[102] R. Mahajan, M. Castro, and A. Rowstron. Controlling the Cost of Reliability in Peer-to-Peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 21–32, Berkeley, CA, USA, 2003. Springer-Verlag.

[103] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC'02)*, New York, NY, USA, 2002. ACM Press.

[104] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed Hashing in a Small World. In *Proceedings of the 4th USENIX*

Symposium on Internet Technologies and Systems (USITS'03), Seattle, WA, USA, March 2003. USENIX.

[105] L. Massoulié, E. Le Merrer, A. Kermarrec, and A. J. Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *Proc. of the 25th Annual ACM Symp. on Principles of Distributed Computing (PODC)*, pages 123–132, 2006.

[106] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR metric. In *Proceedings of the First Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, Lecture Notes in Computer Science (LNCS), pages 53–65, London, UK, 2002. Springer-Verlag.

[107] E. Le Merrer, A.-M Kermarrec, and L. Massoulie. Peer to peer size estimation in large and dynamic networks: A comparative study. In *Proc. of the 15th IEEE Symposium on High Performance Distributed Computing*, pages 7–17. IEEE, 2006.

[108] A. Mislove, A. Post, A. Haeberlen, and P. Druschel. Experiences in building and operating ePOST, a reliable peer-to-peer application. In Willy Zwaenepoel, editor, *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM European Chapter, April 2006.

[109] A. Montresor, M. Jelasity, and Ö. Babaoglu. Chord on Demand. In *Proceedings of the 5th International Conference on Peer-To-Peer Computing (P2P'05)*. IEEE Computer Society, August 2005.

[110] M. Amir Moulavi, Ahmad Al-Shishtawy, and Vladimir Vlassov. State-space feedback control for elastic distributed storage in a cloud environment. In *The Eighth International Conference on Autonomic and Autonomous Systems (ICAS)*, St. Maarten, Netherlands Antilles, March 2012.

[111] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: a Fault-tolerant Algorithm for Atomic Mutable DHT Data. Mit technical report, MIT, June 2005.

[112] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. April 2013. http://bitcoin.org/bitcoin.pdf.

[113] Napster. http://www.napster.com, 2006.

[114] R. Narendula, Z. Miklós, and K. Aberer. Towards access control aware p2p data management systems. In *EDBT/ICDT Workshops*, pages 10–17, 2009.

[115] Taiwan Earthquake on December 2006. http://www.pinr.com/report.php?ac=view_report&report_id=602, January 2008.

[116] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[117] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.

[118] V. Paxson. End-to-end routing behavior in the Internet. *IEEE/ACM Transactions on Networking (TON)*, 5(5):601–615, 1997.

[119] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 288–301, New York, NY, USA, 1997. ACM.

[120] B. Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, Feb 1987.

[121] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the 9th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'97)*, pages 311–320, New York, NY, USA, 1997. ACM Press.

[122] PPLive. http://www.pplive.com/, March 2009.

[123] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4:243–254, January 2011.

[124] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM SIGCOMM 2001 Symposium on Communication, Architecture, and Protocols*, pages 161–172, San Diego, CA, U.S.A., August 2001. ACM Press.

[125] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX'04)*, Boston, MA, USA, June 2004. USENIX.

[126] J. Risson, K. Robinson, and T. Moors. Fault tolerant active rings for structured peer-to-peer overlays. *lcn*, 0:18–25, 2005.

[127] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 2nd ACM/IFIP International Conference on Middleware (MIDDLEWARE'01)*, volume 2218 of *Lecture Notes in Computer Science (LNCS)*, pages 329–350, Heidelberg, Germany, November 2001. Springer-Verlag.

[128] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 2nd ACM/IFIP International Conference on Middleware (MIDDLEWARE'01)*, volume 2218 of *Lecture Notes in Computer Science (LNCS)*, pages 329–350, Heidelberg, Germany, November 2001. Springer-Verlag.

[129] Jan Sacha and Jim Dowling. A gradient topology for master-slave replication in peer-to-peer environments. In *Proceedings of the 3rd International VLDB Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P'05)*, pages 86–97, 2005.

[130] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *In Proc. of MMCN*, 2002.

[131] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E Okasaki, E.H. Siegel, and D.C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39, 1990.

[132] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[133] Tallat M. Shafaat, Bilal Ahmad, and Seif Haridi. Id-replication for structured peer-to-peer systems. In *Euro-Par*, pages 364–376, 2012.

[134] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Handling Network Partitions and Mergers in Structured Overlay Networks. In *Proceedings of the 7th International Conference on Peer-to-Peer Comput-*

ing (P2P'07), pages 132–139. IEEE Computer Society, September 2007.

[135] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. A practical approach to network size estimation for structured overlays. In Karin Anna Hummel and James P. G. Sterbenz, editors, *Proceedings of the Third International Workshop on Self-Organizing Systems (IWSOS'08)*, volume 5343 of *Lecture Notes in Computer Science*, pages 71–83. Springer-Verlag, 2008.

[136] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Dealing with network partitions in structured overlay networks. *Peer-to-Peer Networking and Applications (PPNA)*, 2(4):334–347, 2009.

[137] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Managing Network Partitions in Structured P2P Networks. In X. Shen, H. Yu, J. Buford, and M. Akon, editors, *Handbook of Peer-to-Peer Networking*. Springer-Verlag, July 2009.

[138] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Dealing with bootstrapping, maintenance, and network partitions and mergers in structured overlay networks. In *Proceedings of 6th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), France*, pages 149–158. IEEE Computer Society, 2012.

[139] Tallat M. Shafaat, Monika Moser, Ali Ghodsi, Thorsten Schütt, Seif Haridi, and Alexander Reinefeld. On consistency of data in structured overlay networks. In *Proceedings of the 3rd CoreGRID Integration Workshop*, April 2008.

[140] Tallat M. Shafaat, Monika Moser, Ali Ghodsi, Thorsten Schütt, Seif Haridi, and Alexander Reinefeld. Poster: key-based consistency and availability in structured overlay networks. In *Proc. of the 17th IEEE Symposium on High Performance Distributed Computing*, pages 235–236. ACM, 2008.

[141] Tallat M. Shafaat, Monika Moser, Thorsten Schütt, Alexander Reinefeld, Ali Ghodsi, and Seif Haridi. Key-Based Consistency and Availability in Structured Overlay Networks. In *Proceedings of the 3rd International ICST Conference on Scalable Information Systems (Infoscale'08)*. ACM, June 2008.

[142] A. Shaker and D. S. Reeves. Self-Stabilizing Structured Ring Topology P2P Systems. In *Proceedings of the 5th International*

Conference on Peer-To-Peer Computing (P2P'05), pages 39–46. IEEE Computer Society, August 2005.

[143] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association.

[144] SicsSim. http://dks.sics.se/p2p07partition/, January 2008.

[145] A. Singh, B. Gedik, and L. Liu. Agyaat: mutual anonymity over structured p2p networks. *Internet Research*, 16(2):189–212, 2006.

[146] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.

[147] Skype. http://www.skype.com, November 2012.

[148] SopCast. http://www.sopcast.com/, March 2009.

[149] Skype statistics. http://www.telecompaper.com/news/skype-grows-fy-revenues-20-reaches-663-mln-users, November 2012.

[150] Moritz Steiner, Ernst W. Biersack, and Taoufik En-Najjary. Actively monitoring peers in kad. In *Proceedings of the Sixth International workshop on Peer-To-Peer Systems (IPTPS'07)*, 2007.

[151] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. A global view of kad. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, pages 117–122, New York, NY, USA, 2007. ACM.

[152] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001 Symposium on Communication, Architecture, and Protocols*, pages 149–160, San Deigo, CA, August 2001. ACM Press.

[153] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[154] Internet study. http://www.ipoque.com/sites/default/files/mediafiles/documents/internet-study-2008-2009.pdf, November 2012.

[155] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, 2006.

[156] Undersea cable damage 2006. Taiwan earthquake shakes Internet. http://www.theregister.co.uk/2006/12/27/boxing_day_earthquake_taiwan/, April 2013.

[157] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 172–183. ACM Press, December 1995.

[158] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[159] Werner Vogels. Eventually consistent, January 2012. http://www.allthingsdistributed.com/2007/12/eventually_consistent.html.

[160] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2), 2005.

[161] S. Voulgaris and M. van Steen. Epidemic-Style Management of Semantic Overlays for Content-Based Searching. In *Proceedings of the 11th European Conference on Parallel Computing (EUROPAR'05)*. Springer-Verlag, 2005.

[162] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *J. Network Syst. Manage.*, 13(2):197–217, 2005.

[163] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 1–11. USENIX, November 1994.

[164] B. Y. Zhao, L. Huang, S. C. Rhea, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Global-scale Overlay for Rapid Service Deployment. *IEEE Journal on Selected Areas in Communications (JSAC)*, 22(1):41–53, January 2004.

[165] S.Q. Zhuang, D. Geels, I. Stoica, and R.H. Katz. On failure detection algorithms in overlay networks. In *Proc. of INFOCOM*, 2005.

[166] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.