ROYAL INSTITUTE
OF TECHNOLOGY

Distributed k-ary System: Algorithms for Distributed Hash Tables

ALI GHODSI

A Dissertation submitted to
the Royal Institute of Technology (KTH)
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

December 2006

The Royal Institute of Technology (KTH)
School of Information and Communication Technology
Department of Electronic, Computer, and Software Systems
Stockholm, Sweden

TRITA-ICT/ECS AVH 06:09
ISSN 1653-6363
ISRN KTH/ICT/ECS AVH-06/09–SE
and
SICS Dissertation Series 45
ISSN 1101-1335
ISRN SICS-D–45–SE

Abstract

This dissertation presents algorithms for data structures called *distributed hash tables (DHT)* or *structured overlay networks*, which are used to build scalable self-managing distributed systems. The provided algorithms guarantee *lookup consistency* in the presence of dynamism: they guarantee consistent lookup results in the presence of nodes joining and leaving. Similarly, the algorithms guarantee that routing never fails while nodes join and leave. Previous algorithms for lookup consistency either suffer from starvation, do not work in the presence of failures, or lack proof of correctness.

Several group communication algorithms for structured overlay networks are presented. We provide an *overlay broadcast algorithm*, which unlike previous algorithms avoids redundant messages, reaching all nodes in $O(\log n)$ time, while using $O(n)$ messages, where n is the number of nodes in the system. The broadcast algorithm is used to build overlay multicast.

We introduce *bulk operation*, which enables a node to efficiently make multiple lookups or send a message to all nodes in a specified set of identifiers. The algorithm ensures that all specified nodes are reached in $O(\log n)$ time, sending maximum $O(\log n)$ messages per node, regardless of the input size of the bulk operation. Moreover, the algorithm avoids sending redundant messages. Previous approaches required multiple lookups, which consume more messages and can render the initiator a bottleneck. Our algorithms are used in DHT-based storage systems, where nodes can do thousands of lookups to fetch large files. We use the bulk operation algorithm to construct a pseudo-reliable broadcast algorithm. Bulk operations can also be used to implement efficient *range queries*.

Finally, we describe a novel way to place replicas in a DHT, called *symmetric replication*, that enables parallel recursive lookups. Parallel lookups are known to reduce latencies. However, costly iterative lookups have previously been used to do parallel lookups. Moreover, joins or leaves only require exchanging $O(1)$ messages, while other schemes require at least $\log(f)$ messages for a replication degree of f.

The algorithms have been implemented in a middleware called the *Distributed k-ary System* (DKS), which is briefly described.

Key words: distributed hash tables, structured overlay networks, distributed algorithms, distributed systems, group communication, replication

To Neda, Anooshé, Javad, and Nahid

Acknowledgments

I truly feel privileged to have worked under the supervision of my advisor, Professor Seif Haridi. He has an impressive breadth and depth in computer science, which he gladly shares with his students. He also meticulously studied the research problems, and helped with every bit of the research. I am also immensely grateful to Professor Luc Onana Alima, who during my first two years as a doctoral student worked with me side by side and introduced me to the area of distributed computing and distributed hash tables. He also taught me how to write a research paper by carefully walking me through my first one. Together, Seif and Luc deserve most of the credit for the work on the DKS system, which this dissertation is based on.

During the year 2006, I had the pleasure to work with Professor Roland Yap from the National University of Singapore. I would like to thank him for all the discussions and detailed readings of this dissertation.

I would also like to thank Professor Bernardo Huberman at HP Labs Palo Alto, who let me work on this dissertation while staying with his group during the summer of 2006.

During my doctoral studies, I am happy to have worked with Sameh El-Ansary, who contributed to many of the algorithms and papers on DKS. I would also like to thank Joe Armstrong, Per Brand, Frej Drejhammar, Erik Klintskog, Janusz Launberg, and Babak Sadighi for the many fruitful and enlightening discussions in the stimulating environment provided by SICS.

I would like to show my gratitude to those who read and commented on drafts of this dissertation: Professor Rassul Ayani, Sverker Janson, Johan Montelius, Vicki Carleson, and Professor Vladimir Vlassov. In particular, I thank Cosmin Arad who took time to give detailed comments on the whole dissertation. I also thank Professor Christian Schulte for making me realize, in the eleventh hour, that my first chapter needed to be rewritten. I acknowledge the help and support given to me by the director of graduate studies, Professor Robert Rönngren and the Prefekt, Thomas Sjöland.

Finally, I take this opportunity to show my deepest gratitude to my family. I am eternally grateful to my beloved Neda Kerimi, for always showing endless love and patience during good times and bad times. I also would like to express my profound gratitude to my dear sister Anooshé, and my parents, Javad and Nahid, for their continuous support and encouragement.

Contents

List of Figures

List of Algorithms

1 Introduction

Many organizations and companies are facing the challenge of simultaneously providing an IT service to millions of users. A few search engines are enabling millions of users to search the Web for information. Every time a user types the name of an Internet host, the computer uses the global domain name system (DNS) to find the Internet address of that host. New versions of popular software is sometimes downloaded by millions of users from a single Web site.

The provision of services, such as the ones mentioned above, has many challenges. In particular, the system which provides such large-scale services needs to have several essential properties. First, the design needs to be *scalable*, not relying on single points of failure and bottlenecks. Second, a large-scale system needs to be *self-managing*, as new servers are constantly being added and removed from the system. Third, the system needs to be *fault-tolerant*, as the larger the system, the higher the probability that a failure occurs in some component.

The topic of this dissertation is a data structure called *distributed hash table* (DHT), which encompasses many of the above mentioned properties. This chapter first gives a broad overview of DHTs and their uses. The aim is to motivate the topic, and hence focus on *what* the essential properties and applications are, rather than *how* they can be achieved or built. Thereafter, the contributions of this dissertation are detailed and put in the context of related work. Finally, the organization of the dissertation is presented.

1.1 What is a Distributed Hash Table?

A distributed hash table is, as its name suggests, a hash table which is distributed among a set of cooperating computers, which we refer to as

nodes. Just like a hash table, it contains key/value pairs, which we refer to as *items*. The main service provided by a DHT is the *lookup operation*, which returns the value associated with any given key. In the typical usage scenario, a client has a key for which it wishes to find the associated value. Thereby, the client provides the key to *any* one of the nodes, which then performs the lookup operation and returns the value associated with the provided key. Similarly, a DHT also has operations for managing items, such as inserting and deleting items.

The representation of the key/value pairs can be arbitrary. For example, the key can be a string or an object. Similarly, the value can be a string, a number, or some binary representation of an arbitrary object. The actual representation will depend on the particular application.

An important property of DHTs is that they can efficiently handle large amounts of data items. Furthermore, the number of cooperating nodes might be very large, ranging from a few nodes to many thousands or millions in theory[1]. Because of limited storage/memory capacity and the cost of inserting and updating items, it is infeasible for each node to locally store every item. Therefore, each node is *responsible* for part of the items, which it stores locally.

As we mentioned, every node should be able to lookup the value associated with any key. Since all items are not stored at every node, requests are routed whenever a node receives a request that it is not responsible for. For this purpose, each node has a *routing table* that contains pointers to other nodes, known as the node's *neighbors*. Hence, a query is routed through the neighbors such that it eventually reaches the node responsible for the provided key. Figure 1.1 illustrates a DHT which maps file names to the URLs representing the current location of the files.

Overlay Networks A DHT is said to construct an *overlay network*, because its nodes are connected to each other over an existing network, such as the Internet, which the overlay uses to provide its own routing functionality. The existing network is then referred to as the *underlay network*. If the underlay network is the Internet, the overlay routes requests between the nodes of the DHT, and each such reroute passes through the routers and switches which form the underlay. Overlay networks are

[1]Even though DHTs have never been deployed on such large scale, their properties scale with the system size (see Section 1.3).

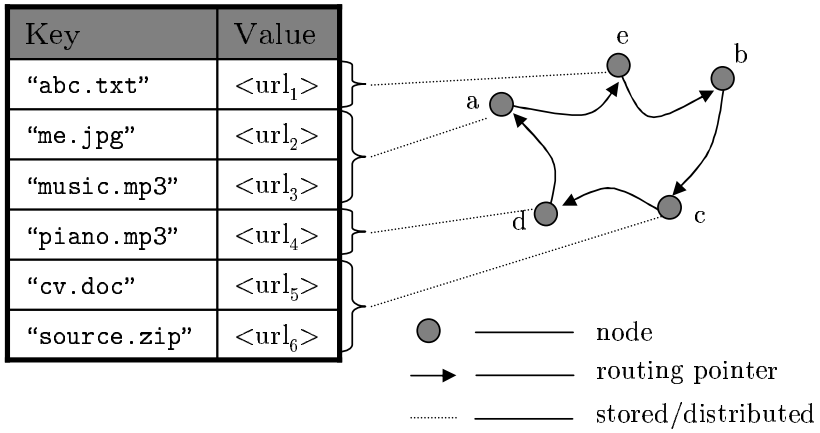

Figure 1.1: Example of a DHT mapping filenames to the URLs, which represent the current location of the files. The items of the DHT are distributed to the nodes a, b, c, d, and e, and the nodes keep routing pointers to each other. If an application makes a lookup request to node d to find out the current location of the file `abc.txt`, d will route the request to node a, which will route the request to node e, which can answer the request since it knows the URL associated with key `abc.txt`. Note that not every node needs to store items, e.g. node b.

also used in other contexts as well, such as for building virtual private networks (VPN). The term *structured* overlay network is therefore used to distinguish overlay networks created by DHTs from other overlay networks. Figure 1.2 illustrates an overlay network and its corresponding underlay network.

There have recently been attempts to build overlays that use an underlay that provides much less services than the Internet. ROFL [21] replaces the underlying routing services of the Internet with that of a DHT, while VRR [20] takes a similar approach for wireless networks.

History of DHTs The first DHTs appeared in 2001, and build on one of two ideas published in 1997:

- *Consistent Hashing*, which is a hashing scheme for caching web pages at multiple nodes, such that the number of cache items needed to be reshuffled is minimized when nodes are added or removed [85, 73].

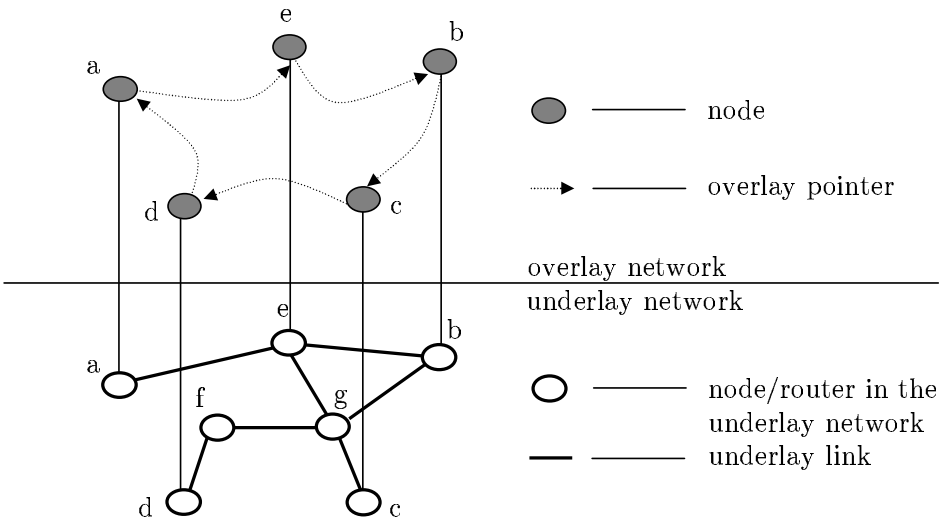

Figure 1.2: An overlay network and the underlay network on top of which the overlay network is built. Messages between the nodes in the overlay network logically follow the ring topology of the overlay network, but physically pass through the links and routers that form the underlay network.

- *PRR*[2] or Plaxton Mesh, which is a scheme that enables efficient routing to the node responsible for a given object, while requiring a small routing table [113].

Of the initial DHTs, Chord [136] builds on consistent hashing, but replaces global information at each node with a small routing table and provides an efficient routing algorithm. Chord has influenced the design of many other DHTs, such as Koorde [72], EpiChord [83], Chord# [127], and the Distributed *k*-ary System (DKS) [5], which this dissertation builds on.

Similarly, PRR is the basis of the initial DHTs Tapestry [143] and Pastry [123]. These systems extend the PRR scheme such that it works while nodes are joining, leaving, and failing.

Content-Addressable Networks (CAN) [116] and P-Grid [2] do not directly build on any of these ideas, though the latter has some resemblance to the PRR scheme.

[2]PRR is derived from the names of the authors – Plaxton, Rajaraman, Richa — who proposed the scheme [113].

Distinguishing Features of DHTs So far, the description of a DHT is similar to the domain name system, which allows clients to query any DNS server for the IP address associated with a given host name. DHTs can be used to provide such a service. There are several such proposals [140, 14], and it has been evaluated experimentally. The initial experiments showed poor performance [32], while recent attempts using aggressive replication, yield better performance results than traditional DNS [114]. Nevertheless, DHTs have properties which distinguish them from the ordinary DNS system.

The property that distinguishes a DHT from DNS, is that the organization of its data is self-managing. DNS' internal structure is to a large extent configured manually. DNS forms a tree hierarchy, which is divided into zones. The servers in each zone are responsible for a region of the name space. For example, the servers in a particular zone might be responsible for all domain names ending with `.com`. The servers responsible for those names either locally store the mapping to IP addresses, or split the zone further into different zones and delegate the zones to other servers. For example, the `.com` zone might contain servers which are responsible for locally storing mappings for names ending with `abcd.com`, and delegating any other queries to another zone. The whole structure of this tree is constructed manually.

DHTs, in contrast to DNS, dynamically decide which node is responsible for which items. If the nodes currently responsible for certain items are removed from the system, the DHT self-manages by giving other nodes the responsibility over those items. Thus, nodes can continuously *join* and *leave* the system. The DHT will ensure that the routing tables are updated, and items are redistributed, such that the basic operations still work. This joining or leaving of nodes is referred to as *churn* or *network dynamism*.

As a side note, it is sometimes argued that a distinguishing feature of DHTs is that they are completely decentralized, while DNS and other systems form a hierarchy, in which some nodes have a more central role than others. However, even though many of the early DHTs are completely decentralized — such as Chord [136], CAN [116], Pastry [123], P-Grid [2], and Tapestry [143] — others are not. Hence, it is more correct to say DHTs are never centralized. In fact, some of the early systems — such as Pastry [123], P-Grid [2], and Tapestry [143] — have an internal design which is flexible. In practice, a minority of the nodes tend to appear

more frequently in routing tables, and hence those nodes will be routed through more often than others. In few of the systems, such as Viceroy [98] and Koorde [72], the design inherently leads to some nodes receiving more queries than others. In summary, the distinguishing feature of DHTs is not complete decentralization, even though they are, to a varying degree, decentralized.

Another key feature of DHTs is that they are fault-tolerant. This implies that lookups should be possible even if some nodes fail. This is typically achieved by replicating items. Hence failures can be tolerated to a certain degree as long as there are some replicas of the items on some alive nodes. Again, as opposed to other systems, such as DNS, fault-tolerance and the accompanying replication are self-managed by the system. This means that the system will automatically ensure that whenever a node fails, some other node actively starts replicating the items of the failed node to restore the replication degree [25, 51].

1.2 Efficiency of DHTs

The efficiency of DHTs has been studied from different perspectives. We mention a few here.

1.2.1 Number of Hops and Routing Table Size

A central research topic since the inception of DHTs has been how to decrease the number of re-routes, often referred to as *hops*, that any given query would take before reaching the responsible node. The reason for this is twofold. First, the latency of transmitting messages is high relative to making local computations. Consequently, removing a hop generally reduces the time it takes to make a lookup. Second, the more hops, the higher the probability that some of the nodes fail during the lookup.

Much research has also been conducted on reducing the size of the routing tables. The main motivation for this has been that the entries in the routing table need to be maintained as nodes join and leave the system. This is referred to as *topology maintenance*. Often, this is done by eagerly probing the nodes in the routing table at regular time intervals to ensure that the routing information is up-to-date [136]. However, lazy approaches to topology maintenance also exist, whereby nodes are added

or removed from the routing table whenever new or failed nodes are discovered [3]. Generally, the bigger the routing table, the more bandwidth is needed to maintain it. Indeed, much theoretical work has been done to find the amount of topology maintenance needed to sustain a working system [97, 77].

There is a trade-off between the maximum number of hops and the size of the routing tables [142]. In general, the larger the routing table, the fewer the number of hops, and vice versa.

Several DHTs [136, 116, 123, 2, 143, 65, 127] guarantee to find an item in hops less than, or equal to, the logarithm of the number of nodes. For example, a system containing 1024 nodes would require maximum $\log_2(1024) = 10$ hops to reach the destination. At the same time, each node would need to store a routing table of size which is logarithmic to the number of nodes.

In many systems [123, 143, 65], the base of the logarithm can be configured as a system parameter. The higher the base, the bigger the routing table and the fewer the hops, and vice versa. In all the PRR-based systems, the routing table size will be $k \cdot L$, where k is the base minus one, and L is the logarithm of the system size with base k. For example, if the base is set to 2, the maximum number of hops in a 4096 node system would be $\log_2(4096) = 12$, while its routing table size would be $1 \cdot \log_2(4096) = 12$. Increasing the base to 16, the maximum number of hops in a 4096 node system would be $\log_{16}(4096) = 3$, while the routing table size would be $15 \cdot \log_{16}(4096) = 45$. Chord has k fixed to 2, while DKS provides a generalization of Chord to achieve any k.

As a side note, we mention two interesting cases as it comes to configuring the base. One is to set the base to the square root of the system size. Then every query can be resolved in maximum two hops. This can be seen by the following equation, when n is set to the number of nodes in the system:

$$\log_{\sqrt{n}}(n) = \log_{\sqrt{n}}((\sqrt{n})^2) = 2$$

The above setting of square root routing tables and two hop lookup is the fixed setting in systems such as Kelips [59] and Tulip [4]. The extreme is to set the base to n, in which case every query can be resolved in one hop, since $\log_n(n) = 1$.

So far, we have mentioned systems in which the routing table size

grows as the number of nodes increases. Nevertheless, systems such as CAN [116] have a constant size routing table. The maximum number of hops will then be in the order of square root of the number of nodes.

Some systems [99, 16, 53, 86] build on the *small worlds* model developed by Kleinberg [75]. This model is influenced by the experiment done by Milgram [102], which demonstrated that any two persons in the USA are likely to be linked by a chain of less than six acquaintances. They guarantee that any destination is asymptotically reached in $\log(n)^2$ hops on average with constant size routing tables. An advantage of the small world DHTs is that they provide flexibility in choosing neighbors.

An question is how much it is possible to decrease the maximum number of hops for a given routing table size. A well known result from graph theory known as the Moore bound [103] gives the optimal number of maximum hops an n node system can guarantee if each node has $\log(n)$ routing pointers. It states that with n nodes, where each node has $\log(n)$ routing entries, the maximum number of hops provided by any system cannot be asymptotically less than $\frac{\log(n)}{\log(\log(n))}$. Some systems, such as Koorde [72] and Distance Halving [108], can indeed guarantee a maximum of $\frac{\log(n)}{\log(\log(n))}$ hops with $\log(n)$ routing pointers [92]. While the design of these systems is intricate, a simpler approach has been suggested for achieving the same bounds. If each node in addition knows its neighbors' routing tables, optimal number of hops can be achieved in many existing DHTs [100, 109]. Note that topology maintenance is avoided for the additional routing tables.

1.2.2 Routing Latency

The number of hops does not solely determine the time it takes to reach the destination, network latencies and relative node speeds also matter. A simple illustrative scenario is a two hop system which routes a message from Europe to Japan and back, just to find that the destination node is present on the same local area network as the source. For another example, consider routing from node d to node e on the ring overlay depicted by Figure 1.2. It takes two hops on the overlay to pass through the path $d - a - e$. But on the underlay it is traveling five hops through the path $d - f - g - e - a - e$.

A metric called *stretch* is often used to emphasize the latency overhead

of DHTs. The stretch of a route is the the time it takes for the DHT to route through that route, divided by the time it takes for the source and the destination to directly communicate. To be more precise, if a lookup in the DHT traverses the hosts $x_1, x_2, \cdots, x_n$, and $d(x_i, x_j)$ denotes the time it takes to send a message from x_i to x_j, then the stretch of that route is $\frac{d(x_1,x_2)+\cdots+d(x_{n-1},x_n)}{d(x_1,x_n)}$. The stretch of the whole system is the maximum stretch for any route. In essence, we are comparing the time it takes for the DHT to route a message through different nodes, with the time it would have taken if the source and the destination had communicated directly without the involvement of a DHT. Notice that in practice, the source and the destination are not aware of each other, since each node only knows a fraction of the other nodes. In fact, in related work called Resilient Overlay Networks [11], it was shown that it might happen that the source and the destination nodes cannot directly communicate with each other on the Internet. But the route that the overlay takes makes communication possible between the two hosts.

Some DHTs, such as the ones based on PRR, are structured such that there is some flexibility in choosing among the nodes in the routing table [123, 143, 2]. Hence, each node tries to have nodes in its routing table to which it has low latency. This is often referred to as *proximity neighbor selection* (PNS). Other systems do not have this flexibility, but instead aim at increasing the size of the routing tables to have many nodes to choose from when routing. This technique is referred to as *proximity route selection* (PRS). Experiments have shown that PNS gives a lower stretch than PRS [58].

As the number of nodes increases, it becomes non-trivial for each node to find the nodes to which it the has the lowest latency. The reason for this is that the node needs to empirically probe many nodes before it finds the closest ones. Work on *network embedding* shows how this can be done efficiently [128]. For example, in Vivaldi [31], each node collects latency information from a few other hosts and thereafter every node receives a coordinate position in a logical coordinate space. For example, in a simple 3-dimensional space, every node would receive a synthetic `(x,y,z)` coordinate. These coordinates are picked such that the Euclidean distance between two nodes' synthetic coordinates estimates the network latency between the two nodes. The advantage of this is that a node does not need to directly communicate with another node to know its latency

to it, but can estimate the latency from the synthetic coordinates of the node, which it can get from other nodes or from a service.

Closely related to latencies are two properties called *content locality* and *path locality*. Content locality means that data that is inserted by nodes within an organization, confined to a local area network, should be stored physically within that organization. Path locality means that queries for items which are available within an organization should not be routed to nodes outside the organization. These two properties are useful for several reasons. First, latencies are lowered, as latencies are typically low within a LAN. The percentage of requests that can be satisfied locally depends on user behavior. But studies indicate that over 80% of requests in popular peer-to-peer applications can be found on the LAN [57]. Second, network partitions and problems of connectivity do not affect queries to data available on the LAN. Third, the locality properties can be advantageous from a security or judicial point of view. SkipNet [65] was the first DHT to have these two properties.

1.3 Properties of DHTs

We briefly summarize the essential properties that most DHTs possess[3]

DHTs are *scalable* because:

- Routing is scalable. The typical number of hops required to find an item is less or equal than $\log(n)$ and each node stores $\log(n)$ routing entries, for n nodes.
- Items are dispersed evenly. Each node stores on average $\frac{d}{n}$ items, where d is the number of items in the DHT, and n is the number of nodes.
- The system scales with dynamism. Each join/leave of a node requires redistributing on average $\frac{d}{n}$ items, where d is the number of items in the DHT, and n is the number of nodes.

DHTs *self-manage* items and routing information when:

- Nodes join. Routing information is updated to reflect new nodes, and items are redistributed.

[3]The numbers are asymptotic and the Big-Oh function should be applied to them.

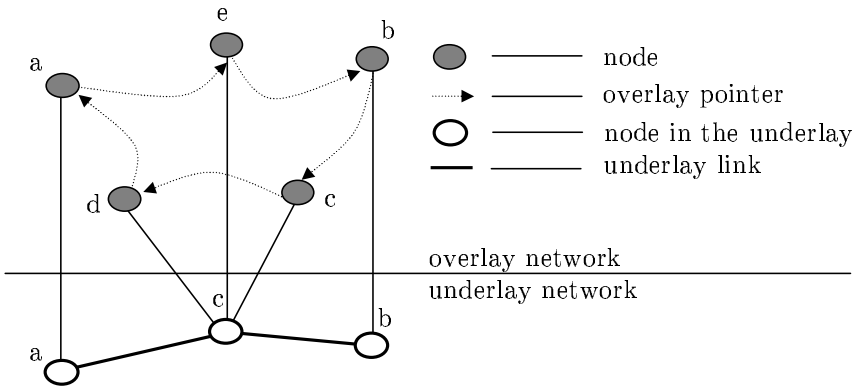

Figure 1.3: A Sybil attack. Node *c* gains majority by imposing as nodes *c*, *d*, and *e* in the overlay network.

- Nodes leave. Routing information is updated to reflect departure of nodes, and items are redistributed before a node leaves.
- Nodes fail. Failures are detected and routing information is repaired to reflect that. Items are automatically replicated to recover from failures.

In addition to the above, some systems self-manage the load on the nodes, while others self-manage to recover from various security threats.

1.4 Security and Trust

Security needs to be considered for every distributed system, and DHTs are no exception. One particular type of attack which has been studied is the *Sybil attack* [39]. The attack is that an adversarial host joins the DHT with multiple identities (see Figure 1.3). Hence, any mechanism which relies on asking several replicas to detect tampered results or detect malicious behavior becomes ineffective. A protection against this is to use some means to establish the true identity of nodes.

One way to establish the identity of the nodes of the DHT is to use public key cryptography. Every node in the DHT is verified to have a valid certificate issued by a trusted certificate authority[4]. Hence, the

[4]It is also possible to use other certificate mechanisms, such as SPKI/SDSI [43], which are based on local knowledge.

nodes in the DHT can be assumed to be trustworthy. This assumption makes sense for certain systems, such as the Grid [47, 119] or a file system running inside an organization. It is, however, infeasible if the system is open to any user, such as an Internet telephony system like Skype.

Establishing node identities using certificates is not sufficient to ensure security. Even trusted nodes can behave maliciously or be compromised by adversaries. Hence, security has to be considered at all levels and the protocols of the system need to be designed such that it is difficult to abuse the system.

Other security issues considered for DHTs include various routing attacks. For example, a node can route to the wrong node, or misinform nodes which are performing topology maintenance. Most of the techniques to prevent these types of attacks involve verifying invariants of the system properties [130], such as ensuring that routing always makes progress toward the destination. Malicious nodes can also deny the existence of data. This can be prevented by comparing results from different replicas, provided that the replicas are not subject to Sybil attacks. Finally, there are DHT specific denial-of-service attacks, such as letting multiple nodes join and leave the system so frequently that the system breaks down [78].

Ultimately, it is impossible to stop nodes from behaving maliciously, especially in a large-scale overlay that is open to any user and does not employ public key cryptography. A key question is then to identify which nodes are trustworthy and which nodes are likely to behave maliciously. One solution to this is to use a node's past behavior and history as an indication of how it will behave in the future. Research on *trust management* aims at doing this by collecting, processing, and disseminating feedback about the past behavior of participating nodes. Despotivić [36] provides a comprehensive survey of the work in this area.

1.5 Functionality of DHTs

So far we have assumed that the ordinary lookup operation is the main use of DHTs. Nevertheless, many other uses are possible. We mention two other operations: *range queries* and *group communication*.

Range Queries In some applications, it might be useful to ask the DHT to find values associated to all keys in a numerical or an alphabetical range. For example, in a grid computing environment, the keys in a DHT can represent CPU power . Hence, an application might query a DHT to search for all keys in the interval $2000 - 5000$ MHz. Range queries in DHTs were first proposed by Andrzejak and Xu [12]. Straightforward approaches to implement range queries in most DHTs are proposed by Triantafillou *et al.* [138] and Chawathe *et al.* [28]. Most such schemes can lead to load imbalance, i.e. that some nodes have to store more items than others. Mercury and SkipNet facilitate range queries without problems of load imbalance [65, 16]. Our work on bulk operations (Chapter 5) can be used in conjunction with most of these systems to make range queries more efficient.

Group Communication The routing information which exists in DHTs can be used for group communication. This is a dual use of DHTs, whereby they are not really used to do lookups for items, but rather just used to facilitate group communication among many hosts. For instance, the routing tables in the DHT can be used to broadcast a message from one node to every other node in the overlay network [42, 49, 118]. The advantage of this is that every node gets the message in few time steps, while every node only needs to forward the message to a few other nodes.

The motivation for doing group communication on top of structured overlay networks is related to Internet's rudimentary support for group communication: IP multicast. Unfortunately, IP multicast is disabled in many routers, and therefore IP multicast often does not work over wide geographic areas. To rectify the situation, early overlay networks such as Multicast Backbone (MBONE) [44] have been used since the inception of IP multicast. The overlay nodes are placed in areas where there is no support for IP multicast. Each node carries a routing table, pointing to other such overlay nodes. These routing tables are then used to connect areas which have no multicast connection between them. Since DHTs have desirable self-managing properties, they have been used, in a similar manner, to enable global multicast. We present one such solution in Chapter 5.

1.6 Applications on top of DHTs

We have now described what a DHT is and overviewed the main strands of research on DHTs. In this section we turn to applications that use DHTs. Our goal is not to give a complete survey of all applications, but rather to convey the main ideas behind the use of DHTs.

1.6.1 Storage Systems

Among the first DHT applications are distributed storage systems. In some systems such as PAST [124], each file to be archived is stored in the DHT under a key which is the hash of the file name, and the value is the contents of the file. The hash of the file name is simply a large integer which is returned when applying a hash function, such as SHA-1, to the filename. Since PAST associates keys with whole files, each node has to store the complete file for each key it is responsible for. If a node does not have enough space, a non-DHT mechanism is used to divert responsibility to other nodes. Popular files are cached along the overlay route to the node on which they are stored. PAST uses public key cryptography together with smart cards to prevent Sybil attacks.

In other systems, such as CFS [33] and our system Keso [10], the concept of *content hashing* is used. A content hash closely relates the key and the value of an item. The key of any item is the hash value of its value. The advantage of this is that once an item is retrieved from the DHT, it can be verified if it has been changed or tampered with by asserting that its key is equal to the hash of the value. Content hashing can be used in conjunction with caching, in which case the self-certifying property of the content-hash makes cache invalidation unnecessary.

CFS stores a whole directory structure in the DHT. Files in CFS are split into smaller *chunks*, which are stored in the DHT using content hashing. The keys of all the blocks belonging to a single file are stored together as an item in the DHT using content hashing. This item is referred to as an *inode* for the file. Hence, each file has an inode item in the DHT, whose value is a set of keys. For each of those keys an item exists in the DHT, whose values are the blocks of the file. Each directory is represented by a *directory block*, whose key is a content hash, and its value is the set of keys of all inodes and directory blocks in the directory. The root directory is also a directory block, but its key is the public key of the node that owns

the directory structure. Hence, to find a file called `/home/user/abc.txt`, the public key of the owner is used to find the root directory block, which should contain the key to the directory block `home`. The directory block for `home` contains the key to the directory block `user`, which contains the key to the inode for the file `abc.txt`. The inode of `abc.txt` contains keys to all chunks, which can be fetched in parallel to reassemble the file [5]. Caching eventually relieves all the lookups made to fetch popular files.

Not all storage systems store the files in the DHT. In fact, it has been shown that beyond a certain threshold, it becomes infeasible to store large amounts of data in a DHT as the number of joins and leaves becomes high [18]. The reason for this is, intuitively, that it takes too long for a node to fetch or transfer the items it is responsible for when it joins and leaves. This has led several storage systems, such as PeerStore [81] and our MyriadStore system [132], to use the DHT for only storing meta data and location information about files.

In summary, DHTs have been used as a building block for many storage systems. The main advantages have been their scalability and self-management properties.

1.6.2 Host Discovery and Mobility

DHTs can be used for host discovery or to support mobility. For example, a node might be assigned dynamic IP addresses, or acquire a new IP address as the result of changing geographic location. To enable the node to announce its new address to any potential future interested parties, the node simply puts an item in the DHT, with the key being a logical name representing the node, and the value being its current address information. Whenever the node changes IP address, it updates its address information in the DHT. Other hosts that wish to communicate with it can find out the node's current address information by looking up its name in a DHT. This is how mobility is achieved in the Internet Indirection Infrastructure (i3) [133].

The above use of DHTs can be found in many projects and several standardization efforts. For example, Host Identity Payload (HIP) [112] aims at separating the names used when routing on the networking layer

[5] The bulk operations introduced in Chapter 5 can be used to do the parallel fetching efficiently.

from the names used between end-hosts on the transport layer. Currently, IP addresses are used for both purposes. HIP proposes replacing the end-host names with a different scheme. A node could then change IP address, which is significant when routing, but keep the same end-host name. To find an end-host's current IP address, a scheme like i3 is proposed to be used. Other similar approaches have been proposed to decouple the two name spaces. For example, in P6P [144], end-hosts use IPv6 addresses, while the core routers in the Internet use IPv4 addresses for routing. Another project, P2PSIP [111], uses a DHT in a similar manner to discover other user agents when initiating sessions for Internet telephony.

1.6.3 Web Caching and Web Servers

Squirrel [69] uses a DHT to implement a decentralized Web proxy. In its simplest form, workstations in an organization form the nodes of a DHT. The Web browsers are configured to use a local program as a proxy server. Whenever the user requests to view a web page, the proxy makes a lookup for the hash of the URL. Initially, the cache will be empty, in which case Squirrel will fetch the requested page from a remote Web server and put it in the DHT, using the hash of the URL as a key, and the contents of the requested page as a value. Hence, Web pages are cached in the DHT. Instead of using a central Web proxy, as many organizations do, a decentralized cache is used based on DHTs.

Another approach is taken by us in DKS Organized Hosting (DOH) [71]. In DOH a group of Web servers form the nodes that make a DHT. Web pages are stored in the DHT, similarly to Squirrel. Some care is taken, however, to ensure that objects related to the same Web page end up having the same key, such that the same node can serve all requests related to the same Web page.

1.6.4 Other uses of DHTs

DHTs have been used in many other contexts, which we mention briefly.

Some relational database systems, such as PIER [67, 93], utilize DHTs to provide scalability, in terms of the number of nodes, which surpasses today's distributed database systems at the cost of sacrificing data consistency.

Many publish/subscribe systems use DHTs. For example, FeedTree [126] is built on top of a DHT to disseminate news feeds (RSS) to clients in a scalable manner. ePOST [106], is a cooperative and secure e-mail system which is built on top of POST [104], which uses a DHT. UsenetDHT [129] provides news-server functionality by storing the contents of the articles in a DHT.

A number of peer-to-peer applications make use of DHTs. Many file sharing applications, such as BitTorrent [30], Azureus, eMule, and eDonkey use the Kademlia DHT [101]. Some systems, such as AP3 [105] and Achord [66], use the DHT as a basic service to provide anonymous messaging or censorship-resistant publishing.

1.7 Contributions

The author is one of the main designers and implementors of a DHT called *Distributed k-ary System* (DKS) and several applications built on top of DKS. He has co-authored the following publications that are related to this research [1, 6, 7, 8, 49, 50, 51, 71, 131, 132]. Rather than describing the full DKS system, we focus on the following contributions: *lookup consistency*, *group communication*, *bulk operations*, and *replication*.

1.7.1 Lookup Consistency

Most DHTs construct a *ring* by assigning an identifier to each node and make nodes point to each other to form a sorted linked list, with its head and tail pointing to each other [136, 72, 65, 123, 143, 98, 16, 83, 61, 122].

We provide algorithms to maintain a ring structure which guarantees *atomic* or *consistent* lookup results in the presence of joins and leaves, regardless of where the lookup is initiated. Put differently, it is guaranteed that lookup results will be the same as if no joins or leaves took place. Second, no routing failures can occur as nodes are joining and leaving. Third, there is no bound on the number of nodes that may simultaneously join or leave the system. Fourth, the provided algorithms do not depend on any particular replication method, and hence give a degree of freedom to the type of replication used in the system. The correctness of all the provided algorithms is proven. Furthermore, we show how ring maintenance can be augmented to handle arbitrary additional routing pointers.

Consequently, lookup consistency is extended to rings with additional pointers, and it is guaranteed that no routing failures occur as nodes are joining and leaving. We show how the algorithms are extended to recover from node failures. Failures only temporarily affect lookup consistency. All algorithms in the dissertation take advantage of lookup consistency.

Related Work

Li, Misra, and Plaxton [89, 88, 87] independently discovered a similar approach to ours. The advantage of their work is that they use assertional reasoning to prove the safety of their algorithms, and hence have proofs that are easier to verify. Consequently, their focus has mostly been on the theoretical aspects of this problem. Hence, they assume a fault-free environment. They do not use their algorithms to provide lookup consistency. Furthermore, they cannot guarantee liveness, as their algorithms are not starvation-free.

In a position paper, Lynch, Malkhi, and Ratajczak [95] proposed for the first time to provide atomic access to data in a DHT. They provide an algorithm in the appendix of the paper for achieving this, but give no proof of its correctness. In the end of their paper they indicate that work is in progress toward providing a full algorithm, which can also deal with failures. One of the co-authors, however, has informed us that they have not continued this work. Our work can be seen as a continuation of theirs. Moreover, as Li *et al.* point out, Lynch *et al.*'s algorithm does not work for both joins and leaves, and a message may be sent to a process that has already left the network [89].

1.7.2 Group Communication

We provide algorithms for efficiently broadcasting a message to all nodes in a ring-based overlay network in $O(\log n)$ time steps using n overlay messages, where n is the number of nodes in the system. We show how the algorithms can be used to do overlay multicast.

Related Work

Previous work done on broadcasting in overlay networks [42] does not work in the presence of dynamism, unlike the algorithms we provide.

Our overlay multicast has several advantages compared to other structured overlay multicast solutions. First, only nodes involved in a multicast group receive and forward messages sent to that group, which is not the case in some other systems [24, 74]. Second, the multicast algorithms ensure that no redundant messages are ever sent, which is not the case with many other approaches [118, 76]. Finally, the system integrates with the IP multicast provided by the Internet.

1.7.3 Bulk Operations

We introduce a new DHT operation called *bulk operation*. It enables a node to efficiently make multiple lookups or send a message to all nodes in a range of identifiers. The algorithm will reach all specified nodes in $O(\log n)$ time steps and it will send maximum n messages, and maximum $O(\log n)$ messages per node, regardless of the input size of the bulk operation. Furthermore, no redundant messages are sent.

We are not aware of any related work, but our bulk operation has been used in several contexts. It is used in DHT-based storage systems [132], where a node might need thousands of lookups to fetch a large file. We use the bulk operation algorithm to construct a pseudo-reliable broadcast algorithm which repeatedly uses the bulk operation to cover remaining intervals after failures. Finally, the algorithms are used to do replication in Chapter 6 and by some of the topology maintenance algorithms [50].

1.7.4 Replication

We describe a novel way to place replicas in a DHT called *symmetric replication*, which makes it possible to do parallel recursive lookups. Parallel lookups have been shown to reduce latencies [120]. Previously, however, costly iterative lookups have been used to do parallel lookups [120, 101]. Moreover, joins or leaves only require exchanging $O(1)$ message, while other schemes require at least $\log(f)$ messages for a replication degree f. Failures are handled as a special case, which requires a more complicated operation, using more messages.

Related Work

Closest to our symmetric replication is the use of multiple hash functions. Nevertheless, this scheme has one disadvantage. It requires the inverse of the hash functions to be known in order to maintain the replication factor (see Chapter 6). Even if the inverse of the hash functions were available, each single item that the failed node maintained would be dispersed all over the system when using different hash functions, making it necessary to fetch each item from a different node. This is infeasible as the number of items is generally much larger than the number of nodes.

Later, others have rediscovered variations of symmetric replication [84, 64].

1.7.5 Philosophy

Much of the research on DHTs has been done under the wide umbrella of *peer-to-peer computing*. The following quote from the seminal paper on Chord [134, pg 2] motivates this:

> In particular, [Chord] can help avoid single points of failure or control that systems like Napster possess [110], and the lack of scalability that systems like Gnutella display because of their widespread use of broadcasts [54].

A similar quote can be found in the original paper on CAN [117, pg 1].

We believe that one of the main motivational scenarios for DHTs has been a peer-to-peer application that is used by hundreds of thousands of simultaneous desktop users, each being part of the DHT. The vision has been to have an efficient and decentralized replacement for common file-sharing applications. This implicitly carries many assumptions, such as untrusted nodes, high churn, and varying latencies. Most importantly, desktop users can anytime turn their computers off, and hence there is a high frequency of failures. For that reason, failures and leaves can be considered as the same phenomena.

In contrast, our philosophy has been that DHTs are useful data structures, whose applicability is not confined to peer-to-peer applications. They might well be used in a system consisting of a few hundred, or thousand nodes. The nodes in the DHT might be formed by dedicated

servers within one or several organizations, such as in the Grid [47, 119]. Hence, while the system should be fault-tolerant, failures might not be the common case. Similarly, the nodes in the DHT can be equipped with digital certificates, which allow for authentication and authorization. Consequently, the nodes can in general be trusted, provided the right credentials.

Given our philosophy, we have tried to investigate what can be done on DHTs in less harsh environments. Each of the contributions has a direct connection to this philosophy. The lookup consistency algorithms differentiate between leaves and failures, and are able to give strong guarantees while joins and leaves are happening, while failures introduce some uncertainty. The group communication algorithms are suitable for stable environments where their efficiency is advantageous. Their use can, however, be questioned in environments with high failure rates, as the algorithms might never terminate. Our symmetric replication simplifies the handling of joins and leaves by only requiring $O(1)$ messages to transfer replicas. Failures are handled as a special case, which involve a more complicated operation, which requires more messages.

1.8 Organization

The chapters of this dissertation are organized as follows:

- Chapter 2 presents our model of a distributed system. It also presents the event-driven and control-oriented notation that is used throughout the dissertation to describe algorithms. Finally, the chapter presents the Chord system, which the rest of the dissertation assumes as background knowledge.
- Chapter 3 provides algorithms for constructing and maintaining a ring in the presence of joins, leaves, and failures. The algorithms guarantee *atomic* or *consistent* lookups.
- Chapter 4 shows how the ring can be extended with $(k-1)\log(n)$ additional pointers to provide $\log_k(n)$ hop lookups, in an n node system. It provides different routing algorithms and provides efficient mechanisms to maintain the topology up-to-date in the presence of joins, leaves, and failures. Finally, it shows how the additional routing pointers can be maintained to guarantee that there

are no routing failures when nodes are joining and leaving, while providing lookup consistency.

- Chapter 5 provides algorithms for broadcasting a message to all nodes in a ring-based overlay network. Moreover, it shows how the broadcast algorithm can be used to do overlay multicast. Chapter 5 also introduces a new DHT operation called *bulk operation*, which enables a node to efficiently make multiple lookups or send a message to all nodes in a range of identifiers.

- Chapter 6 describes *symmetric replication*, which is a novel way to place replicas in a DHT. This scheme makes it possible to do recursive parallel lookups to decrease latencies and improve load balancing. Another advantage of symmetric replication is that a join or a leave requires the joining or leaving node to exchange data with only one other node prior to joining or leaving.

- Chapter 7 briefly describes the implementation of a middleware called *Distributed k-ary System* (DKS), that implements the algorithms presented in this dissertation.

- Chapter 8 provides a conclusion and points to future research directions for DHTs.

2 Preliminaries

This chapter briefly describes our model of a distributed system. Thereafter, we informally introduce the pseudocode conventions used to describe algorithms. Finally, we describe Chord, which provides a DHT.

2.1 System Model

In this section, we present our model of a distributed system. The system consists of *nodes*, which communicate by *message passing*, i.e. the nodes communicate with each other by sending messages.

We make the following three assumptions about distributed systems, unless stated otherwise:

- *Asynchronous system*. This means that there is no known upper bound on the amount on the time it takes to send a message[1] or to do a local computation on a node.
- *Reliable communication channels*[2]. A channel is reliable if every message sent through it is delivered exactly once, provided that the destination node has not crashed. Moreover, we assume that a node can never receive a message that has never been sent by some node. Hence, there can be no loss, duplication, garbling, or creation of messages.
- *FIFO communication channels*. This means that messages sent on a channel between two nodes are received in the same order that they were sent.

[1] This assumption is sometimes known as *asynchronous network*.

[2] Reliable communication channels are sometimes referred to as perfect communication channels [56, pg 38ff].

The last two properties are already satisfied by the connection-oriented TCP/IP protocol used in the Internet, and can be implemented over unreliable networks by marking packets with unique sequence numbers, using timeouts, packet re-sending, and storage of sequence numbers to filter duplicate messages. For more information on their implementation see Guerraoui and Rondrigues [56, Chapter 2].

2.1.1 Failures

If nothing else is said, we generally assume that there are no failures. We do, however, always consider nodes joining and leaving. Furthermore, all our algorithms are augmented to handle failures. When failures are introduced, we assume that processes can crash at any time, in which case they stop communicating. We will use *unreliable failure detectors* to detect when a node has failed [26]. The algorithms we present have been designed to work on the Internet. Therefore, we only consider failure detectors which are suitable for the Internet. We assume that every failure detector is *strongly complete*, which means that it eventually will detect if a node has crashed. This assumption is justifiable, as it can be implemented by using a timer to detect if some expected message has not arrived within some time bound. Thus, a failure is eventually always detected. A failure detector might, however, be *inaccurate*, which means that it might give false-negatives, suspecting that a correct, albeit slow, node has crashed. If timers are used to implement failure detectors, then inaccuracy stems from timers that expire before the receipt of the corresponding message. Sometimes we need accuracy to ensure the termination of an algorithm. In those cases, we strengthen our assumptions about the asynchrony in the system. We then assume that the failure detector is *eventually strongly accurate*, which means that after some unknown time period, the failure detector will not inaccurately suspect any node as failed. The class of failure detectors referred to as *eventually perfect* are strongly complete and eventually strongly accurate.

2.2 Algorithm Descriptions

Throughout this dissertation, we will use a node's identifier to refer to it, i.e. we will write "node i" instead of "a node with identifier i". We

use pseudocode which resembles the Pascal programming language. The next two sub-sections introduce two different notations that are used in this dissertation.

2.2.1 Event-driven Notation

Most of the message passing algorithms will be described using *event-driven* notation. There is one event handler for each message. The message handler describes the parameters of the message, and the actions to be taken when a message is received. The actions include making local computations, such as updating local variables, and possibly sending messages to other nodes. The advantage of this model is that each node can be modeled as a state-machine, which in each state transition receives a message, updates its local state by doing local computations, and sends zero or more messages to other nodes. Each such transition is sometimes referred to as a *step*.

The following example shows a message handler for the message MessageName1, with parameter $p1$. The handler declares that if a MessageName1 message is received at node n from node m with a parameter $p1$, it should do some local computation and then send a MessageName2 message to p with parameter $p2$. Execution of event handlers is serialized, i.e. a node can only executing at most one event handler at any given point in time. Only one parameter is used in the example, but any number of parameters can be specified by separating them with a comma.

```
1: event n.MessageName1(p1) from m
2:     local computations
3:     sendto p.MessageName2(p2)
4:     local computations
5: end event
```

The event-driven notation assumes asynchronous communication[3]. That means that the sending of a message is not synchronized with the receiver. As a side note, this is the reason why a single state-transition can be used to model the receipt of a message, local computations, and the sending of messages.

[3]Asynchronous communication should not be confused with asynchronous networks

2.2.2 Control-oriented Notation

In some cases, we find it convenient to describe the algorithms in control-oriented notation. In this notation a node can do local computations and then explicitly wait for a message of a particular type. This is called a *blocking receive*. We differentiate blocks of code using control-oriented notation with the keyword *procedure*. In the control-oriented notation, we no longer assume that a node will be executing at most one procedure. A procedure can also return a value, similarly to a function in an ordinary programming language.

The following example declares that if a procedure n.ProcedureName is executed at node n with a parameter $p1$, it should do some local computation, send MessageName1 with parameter $p2$. Thereafter, the computation blocks and waits for the receipt of a MessageName2 message with parameter $p3$ from *any* node m. Note that it waits for the message from any node, and once the message is received the variable m is set to the sending node's identity. Thereafter, the computation blocks waiting for the receipt of a MessageName3 with some parameter $p4$ from the *specified* node i. Local procedure calls do not need the identifier prefix, i.e. $proc()$ denotes making a call to the local procedure $proc()$ at the current node.

```
1: procedure n.ProcedureName(p1)
2:     local computations
3:     sendto p.MessageName1(p2)
4:     receive MessageName2(p3) from m
5:     receive MessageName3(p4) fromthis i
6:     local computations
7: end procedure
```

Note that this notation is not as straight-forward to model with state-machines, as the event-driven notation.

Synchronous Communication It is sometimes convenient to synchronize the sending of a message with the receipt of the message. This can be done by using *synchronous communication*. Note that we still assume an asynchronous network, in which there are no known time bounds on events. Given an asynchronous system, the only way to implement

synchronous communication is by sending a message and waiting for an acknowledgment from the receiver. Since an acknowledgment message must be sent by the receiver for every received message, the receiver can piggy-back parameters on the acknowledgment back to the sender. This corresponds to remote-procedure calls (RPC), where a node can call a procedure at another node and await the result of the execution of the procedure.

Synchronous communication can be implemented using the control-oriented notation we introduced. This can be achieved by always having a blocking receive for an acknowledgment after each send, and correspondingly sending an acknowledgment after each receive event. We will use RPC prefix notation as a shorthand for this. Hence, an expression i.Proc($p1$) means executing the procedure Proc($p1$) at node i and returning its value back to the caller. This is implemented in control-oriented notation by the following:

```
1: procedure n.EmulateRPC()
2:     sendto i.ProcReq(p1)
3:     receive ProcReply(result) fromthis i
4:     return result
5: end procedure

6: event n.ProcReq(p1) from m
7:     res = Proc(p1)                          ▷ Call local procedure
8:     sendto m.ProcReply(res)
9: end event
```

Similarly, we use RPC notation for reading a remote variable. Hence, $i.var$ denotes fetching the value of the variable var at node i. This can be implemented using control-oriented notation by the following:

```
1: procedure n.EmulateRPCGet()
2:     sendto i.VarReq()
3:     receive VarReply(result) fromthis i
4:     return result
5: end procedure

6: event n.VarReq() from m
7:     sendto m.VarReply(var)
8: end event
```

Writing to a remote variable can be implemented in a similar manner.

2.2.3 Algorithm Complexity

The efficiency of our distributed algorithms will be measured in terms of resource consumption and time consumption. We assume that local computations consume negligible resources and take negligible time compared to the overhead of message passing.

We use *message complexity* as a measure of resource consumption. The message complexity of an algorithm is the total number of messages exchanged by the algorithm. Sometimes, the message complexity does not convey the real communication overhead of an algorithm, as the size of the messages is not taken into account. Hence, on a few occasions, we use *bit complexity* to measure the total number of bits used in the messages by some algorithm.

Time complexity will be used to measure the time consumption of an algorithm. We assume that the transmission time takes *at most* one time unit and all other operations take zero time units. The worst case time complexity is often the same if we assume that the transmission of a message takes *exactly* one time unit, but for some algorithms the worst time complexity increases if we assume that the time it takes to send a message takes *at most* one time unit.

Unless specified, we assume that our complexity measures denote the *worst-case* complexity of a given algorithm.

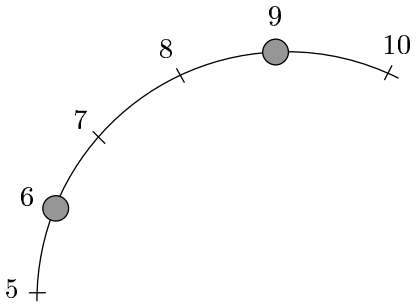

Figure 2.1: Node 9 is responsible for the identifiers between its predecessor, 6, and itself, i.e. the identifiers $\{7, 8, 9\}$.

2.3 A Typical DHT

We briefly describe Chord [134]. The choice of Chord is motivated by it being well known, making it attractive for pedagogical purposes. We first briefly cover the Chord basics. Thereafter we show how Chord handles network dynamism.

Every structured overlay network makes use of an *identifier space*. The identifier space, denoted $\mathcal{I}$, consists of the integers $\{0, 1, \cdots, N-1\}$, where N is some a priori fixed, large, and globally known integer. This identifier space is perceived as a ring that wraps around at $N-1$.

Every node in the system, has a unique identifier from the identifier space. We refer to the set of all nodes present at any given time as $\mathcal{P}$. We currently ignore how a node gets its identifier, but one can imagine that it can randomly pick an identifier from a very large identifier space to ensure the uniqueness of the identifier with high probability. Each node keeps a pointer[4], *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. 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. The successor pointers form a ring, which resembles a "distributed linked list" that is sorted by the identifiers of the nodes and its tail node points to its head node. The predecessors also form such a distributed

[4]By pointer we mean that the node's identifier and network address is stored such that communication can be established with it.

linked list. Hence, the *succ* and *pred* pointers form a distributed circular doubly-linked list. From now on we refer to this distributed structure as a *ring* or a *doubly-linked ring*.

Every identifier in the identifier space is under the responsibility of a node in the following way. The whole identifier space is partitioned into P intervals, where P is the current number of nodes in the system. Each node, n, is *responsible* for one interval. In Chord, a node is responsible for the interval consisting of all identifiers in the range starting from, but excluding its predecessor's identifier up to, and including its own identifier (see Figure 2.1).

2.3.1 Formal Definitions

For preciseness, we include formal definitions of the above descriptions.

We will use the notation $x \oplus y$ for $(x + y)$ modulo N for all $x, y \in \mathcal{I}$, where $N = |\mathcal{I}|$. Similarly, $x \ominus y$ is defined as $(x - y)$ modulo N for all $x, y \in \mathcal{I}$. For example, if the size of the identifier space is 16, then $15 \oplus 2 = 1$, while $1 \ominus 2 = 15$.

Distances on the identifier space are measured in clockwise direction. Hence, the distance d between any two identifiers x and y is defined as:

$$d(x, y) = y \ominus x$$

The *successor of an identifier* is its closest node in clockwise direction. Hence, the successor S of an identifier x for a set of nodes $\mathcal{P}$ is defined as:

$$S(x) = x \oplus \min\{d(x, y) \mid y \in \mathcal{P}\}$$

The *successor of a node* p is therefore defined by the function *succ* at node p as:

$$succ = S(p \oplus 1)$$

Similarly, the *predecessor of a node* p is defined as the node farthest away in clockwise direction. Hence, the predecessor of p is defined by the function *pred* at node p as:

$$pred = p \oplus \max\{d(p, y) \mid y \in \mathcal{P}\}$$

A node p is the responsible for an identifier x if and only if:

$$S(x) = p$$

2.3.2 Interval Notation

We now introduce some notation to make our discussions about the identifiers and intervals on the ring more precise. The whole identifier space can be represented by an interval of the form $[x, x)$ or $(x, x]$ for an arbitrary $x \in \mathcal{I}$, where the start of an interval is excluding the first identifier if the left bracket is round, (, and it is including the first identifier if it is square, [. Similarly, the end of an interval is including the last identifier if the right bracket is square,], and excluding the last identifier if it is round,). For any $x \in \mathcal{I}$, we note that $[x, x] = \{x\}$ and $(x, x) = \mathcal{I} \backslash \{x\}$. Hence, a node n is responsible for $(n.pred, n]$. For example, if the size of the identifier space is 16, then $(2, 10]$ is the set of identifiers $3, 4, \cdots, 9, 10$. The interval $(10, 2]$ is equivalent to the identifiers $0, 1, 2$, and $11, 12, 14, 15$.

Interval Notation and Sets of Identifiers We now connect the interval notation to a set representation. So far we have used the notation of the sort $(i, j]$ to represent intervals of the identifier space. Such an interval is a compact representation of a set of identifiers. For example, in an identifier space of size 16, the interval $(14, 3]$ represents the set of identifiers $\{15, 0, 1, 2, 3\}$. It is therefore possible to apply the operations available for sets on intervals, such as taking the union or intersection of two intervals. For example, the interval $[11, 15]$ represents the set of identifiers $\{11, 12, 13, 14, 15\}$. Therefore, the union of the intervals, $(14, 3] \cup [11, 15]$, is the set of identifiers $\{0, 1, 2, 3, 11, 12, 13, 14, 15\}$. Similarly, the intersection of the intervals, $(14, 3] \cap [11, 15]$, is the set of identifiers $\{15\}$. It might, of course, not be possible to represent such a set of identifiers as a single interval.

In our algorithms, we make extensive use of the basic set operations on intervals. The reason for this is that the semantics of the operations are well defined. In a system implementation, these can be implemented and optimized as fit.

2.3.3 Distributed Hash Tables

A DHT is like an ordinary hash table, except that the key/value pairs in the hash table are distributed and stored among the nodes in the system (see Chapter 1).

The DHT is implemented by deterministically assigning an identifier

to every key/value pair in the DHT using a globally known hash function, H. Specifically, a key value pair $\langle k, v \rangle$ is mapped to the identifier $H(k)$. Each node locally stores the key/value pairs whose identifiers it is responsible for.

Any node can lookup the value associated with any key by making a *lookup*. More precisely, any node can perform a lookup to find out which node is currently *responsible* for a key, and thereafter directly contact that node to find out the value associated with the key. Similarly, a DHT put, delete, or update operation can be implemented by making a lookup for the particular key, and then asking the responsible node to perform the desired operation. The lookup is done by traversing the successor pointers until a node is reached whose successor is responsible for the destination identifier.

For example, the DHT can contain the key/value pair $\langle$"age","old"$\rangle$, which is assigned the identifier $H(\text{"age"}) = 15$. The node which is responsible for the identifier 15 stores this key/value pair locally. All that is needed for a node to find the value associated with "age", is to lookup the node currently responsible for the *destination identifier* 15. The responsible node is then contacted to find out that "old" is the value for the key "age".

2.3.4 Handling Dynamism

When a node joins, or leaves, the system needs to ensure that the ring structure is intact, i.e. that each node is indeed pointing to its correct successor and predecessor.

When a new node joins the system it proceeds in three steps. First, it needs the address of an existing node in the system. Second, it needs to find its successor on the ring. Third, it needs to incorporate itself into the ring, by letting some nodes update their successor and predecessor pointers. We briefly describe each of these three steps.

Finding the address of an existing node is often considered out of the scope of most research papers. We briefly mention three approaches here. One approach is to use a distributed cache server, such as the GWebCache [60]. This is essentially a server that keeps a cache of some nodes that are currently in the system. The server can randomly contact nodes in its cache and query them for more nodes, such that the cache always contains alive nodes. New nodes know the address of one or more dis-

tributed cache servers, which they contact to get a reference to an existing node. Jelasity *et al.* [70] describe how such a *sampling service* can efficiently be implemented. Another approach is to keep a local cache file on each client, which initially contains a predefined set of nodes. Each time a node wants to join, it tries to find an alive node from its local cache file. The local cache is updated with up-to-date information each time the application is used. A third approach is to use IP multicast or broadcast on the local area network to find a node which is already a member of the DHT. In practice, a combination of these three methods is used.

Finding the successor of a new node, n, is trivially achieved by following successor pointers until a node is reached whose *successor* is responsible for the identifier n. This would require $P - 1$ messages in the worst case as the whole ring would need to be traversed, where P is the number of nodes in the system. In practice, a much more efficient search is performed, as we show in Chapter 4.

To ensure that the new node, its predecessor, and its successor, all have correct *succ* and *pred* pointers, Chord uses a *periodic stabilization* algorithm. The algorithm shown in Algorithm 1 is run periodically at each node. Initially, a new node sets its successor pointer to its actual successor on the ring, and its predecessor pointer to itself. The periodic stabilization algorithm will ensure that all nodes eventually correct their successor and predecessor pointers correctly. An example of this is illustrated in Figure 2.2, which shows how stabilization works when a new node joins the system.

Leaves are handled using periodic stabilization in conjunction with a *successor-list*. The successor-list at a node n is just a special routing table with a list of n's closest consecutive successors. The size of the list is some constant. Whenever a node detects that its predecessor has failed, it changes its *pred* pointer to point to itself. Whenever a node detects that its successor has failed, it makes its *succ* pointer point to the next alive node in its successor-list. Hence, if a node fails, its successor q will detect that and sets $q.pred = q$. Furthermore, the failed node's predecessor p will detect the failure, and set $p.succ = q$. The next time p performs periodic stabilization, q will be notified about p, and hence sets $q.pred = p$.

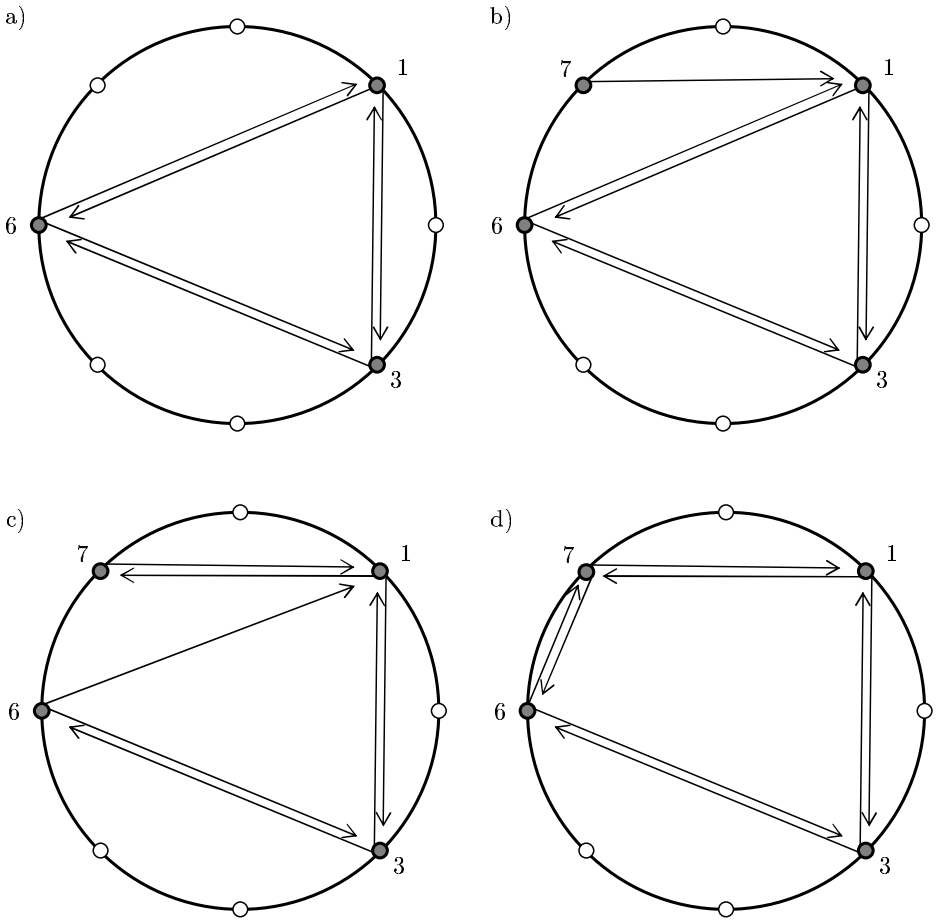

Figure 2.2: a) system with 3 nodes with correct successors and predecessors. b) node 7 joins and sets its successor pointer correctly. c) node 7 runs periodic stabilization. d) node 6 runs periodic stabilization.

Algorithm 1 Chord's periodic stabilization protocol

```
 1: procedure n.STABILIZE()
 2:     p := succ.GetPredecessor()
 3:     if p ∈ (n, succ) then
 4:         succ := p
 5:     end if
 6:     succ.Notify(n)
 7: end procedure

 8: procedure n.GETPREDECESSOR()
 9:     return pred
10: end procedure

11: procedure n.NOTIFY(p)
12:     if p ∈ (pred, n] then
13:         pred := p
14:     end if
15: end procedure
```

3 Atomic Ring Maintenance

In this chapter we explain how the nodes that participate in the overlay form a distributed ring, where each node points to its successor and predecessor in the ring. This basic ring structure is the basis of many structured overlay networks [134, 72, 65, 123, 143, 98, 16, 83, 61, 122], and the rest of the dissertation shows how it can be used to build various services, such as DHTs and group communication services. We show how this ring should be maintained as nodes join and leave. In particular, we guarantee that joins and leaves do not affect the consistency of the results when traversing the successor and predecessor pointers to find the responsible node for an identifier. Furthermore, the algorithms guarantee that a lookup will never be directed to a node that has left the system.

Given our philosophy of DHTs (see Section 1.7.5), we differentiate between node failure and benign departure of nodes. In the former case, a node crashes without synchronization. In the latter case, a node synchronizes its departure with other nodes, prior to leaving. We will from now on refer to the latter case as a node *leaving*.

Our goal is to ensure that joins and leaves do not affect the functionality provided by the rest of the system. For example, we want to ensure that lookups to items in the hash table succeed while nodes are continuously joining and leaving the system. This is non-trivial because the set of nodes present in the system determines which node stores which data item. Hence, any change to the set of present nodes requires movement of data between the nodes. Since lookups can take place concurrently with these changes, the problem becomes intricate.

We introduce *atomic ring maintenance* to ensure correctness of lookups in the presence of joining and leaving nodes. In essence, we serialize interfering joins and leaves, i.e. they are done sequentially, rather than

concurrently, to avoid inconsistencies. Before getting to the details of this, we motivate the need for such algorithms by showing problems which arise in existing systems that do not serialize joins and leaves.

3.1 Problems Due to Dynamism

We now turn to some of the problems that can occur when nodes join and leave the Chord system.

Problems with Joins Imagine a Chord system which contains nodes with identifiers 3 and 9. Initially 3's successor pointer points at 9 and 9's predecessor pointer points at 3. A new node with identifier 7 joins the system (see Figure 3.1a), sets its successor pointer to 9, and performs a periodic stabilization, which results in node 9's predecessor pointer pointing at 7 (Figure 3.1b). Meanwhile, another node 5 joins, sets its successor pointer to node 9 (Figure 3.1c) and performs a periodic stabilization. The pointers in this system are as follows (Figure 3.1d): 3's successor is 9, 5's successor is 7, and 7's successor is 9. At the same time, 9's predecessor is 7, and 7's predecessor is 5. If a lookup to identifier 6 arrives at node 3, node 3 will (according to the description of a lookup) return the address of node 9 as it believes that 9 is responsible for the interval $\{4, \cdots, 9\}$. At the same time, if a lookup for identifier 6 is initiated at node 5, it will respond that the responsible node is 7.

The above scenario is problematic as a lookup can either return node 7 or node 9 as the responsible for identifier 6. Similarly, an update or insertion to node 9 will not have effect as lookups will be directed to node 7 as soon as node 3 stabilizes and points to node 5.

Problems with Leaves In Chord, nodes can leave unnoticed, without synchronizing with any other nodes. The idea is that the periodic stabilization will eventually correct the pointers. Before that happens, the pointers might be in an inconsistent state. Consequently, a lookup might lead to a node that has left the system. In such cases, the node referring to the absent node needs to detect that the node is no longer in the system, using timeouts, so that it can replace that node in its routing table. Only thereafter the lookup can proceed to an alive node. Furthermore, the overall number of leaves that can be tolerated by the system will depend

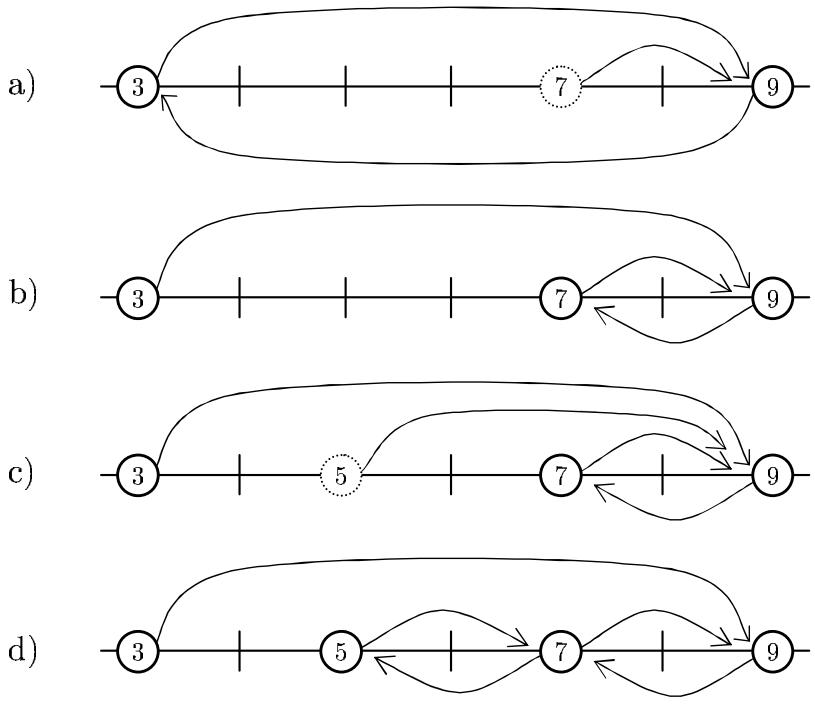

Figure 3.1: Example of inconsistent stabilization.

on the frequency of the stabilization and the size of the successor-list. For if too many adjacent nodes leave between two stabilizations, there might not exist another live node in the successor-list of the node that detects the failure of its successor.

We proceed by a simple example to demonstrate the apparent difficulties in synchronizing a leave to ensure that lookups are unaffected by leaves. As in Figure 3.2, assume node 5 has 10 as its successor, and node 10 has 15 as its successor. If node 10 wants to leave the system, we have to ensure that items stored on node 10 are made available to 15, and ensure that the routing information in the system is updated such that lookups for identifiers $\{6, \cdots, 10\}$ are forwarded to node 15. This requires that 10 stays in the system at least until 5 has updated its successor pointer and until 10's data is made available on 15.

To continue the example, node 15 might want to leave the system at the same time as node 10. Due to the asynchrony in the system, it might be that both 10 and 15 concurrently inform their respective predecessor

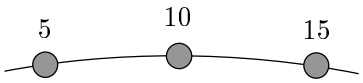

Figure 3.2: Perfect system state before a leave operation.

about their departure and instruct their predecessors to point to their successors. This might result in node 5 pointing to node 15 even though both node 10 and 15 have left the system. Node 5 incorrectly points to 15, and might incorrectly forward lookups to it, leading to a *routing failure*.

The apparent problems which occur due to joins and leaves can be overcome by serializing joins and leaves.

3.2 Concurrency Control

As we mentioned earlier, the aim is to maintain a ring. In non-distributed data structures, the approach often taken is to lock the whole data structure when adding and removing elements from it. Hence, the list is guarded against becoming corrupt due to concurrent modifications. This approach can naïvely be applied to our distributed ring. However, the performance overhead of locking all the nodes becomes large as the size of the ring grows.

Another approach to avoid inconsistencies due to concurrent modifications of the ring is to acquire three locks, one for the predecessor, one for the successor, and one for the joining/leaving node. After acquiring the locks, the pointers of the respective nodes can be updated to allow a node join or leave the ring. Since a join or leave of a node q only requires changes to the pointers of node q, q's predecessor, and q's successor, attempting to lock those three nodes against concurrent modifications would solve concurrency related problems. There is, however, a simpler approach to protect the nodes from concurrent modifications, which is superior to the solution we just described.

Instead of locking the joining/leaving node, as well as its predecessor and successor, we now describe a simpler approach which is reducible to the well known problem of *the dining philosophers* [37], which we explain later.

Assume every node *i hosts* a lock L_i, which can only be acquired by at

most one node. A lock L_i can be acquired by any node, including i itself. We differentiate between a node hosting a lock, and a node holding a lock. The former indicates that the node is responsible for managing the lock, and has nothing to do with whether the lock is free or not. The latter indicates that the node has acquired the lock, which is no longer free.

In our join and leave algorithms, the joining or leaving node n will first acquire its own lock L_n, and thereafter its successor's lock ($L_{n.succ}$). Only once it has acquired both locks, it can update its own pointers, its predecessor's *succ* pointer, and its successor's *pred* pointer. Thereafter it will release both locks. This reduces the number of locks to two, with one of them being a local lock, which can be acquired without the overhead of sending or receiving any messages.

To ensure that this scheme withstands concurrent access, we need to show that this scheme will have the desired *safety* and *liveness* properties. A safety property expresses that something will *not* happen, while a liveness property shows that something *must* happen [79]. In practice, safety properties are used to show that the algorithm never exhibits bad behavior by showing that some undesirable property never happens. Liveness properties are used to show that something good will eventually happen by showing that some desired property must always be true.

We motivate the types of proofs that we provide for the atomic ring maintenance. If the ring was not concurrently modified and traversed, we would be dealing with a data structure which would resemble local linked lists, which can be fully locked and accessed atomically. Such local data structures are commonplace in computing engineering, and a proof of their correctness is not our concern. Instead, we focus on showing that the concurrency related aspects of the algorithms are correct.

3.2.1 Safety

The safety property we want is that a joining or leaving node j will be able to update its own pointers, as well as its predecessor's and successor's, without risking the predecessor or successor leaving the system before the join or leave has completed. Furthermore, the pointers to be updated should not be altered as a consequence of other joins or leaves happening before j has finished updating them.

In the following we will say that the pointers in the system are correct if every node's successor pointer correctly points to its successor, and

every node's predecessor pointer correctly points to its predecessor in the ring.

Next, we prove the safety property that when a joining/leaving node j successfully acquires the necessary locks, it will have mutually exclusive access to the pointers it wants to alter, and the nodes hosting those pointers will remain in the system until j finishes its operation.

Theorem 3.2.1 (Non-interference)**.** *Assume a system, of at least two nodes, with correct pointers. If a node j successfully acquires the locks L_j and $L_{j.succ}$, then j's successor q ($j.succ$) and predecessor p ($j.pred$ if j is leaving, and $j.succ.pred$ if j is joining) cannot leave the system until the locks are released. Furthermore, no other join or leave operation will affect the pointers $p.succ$, $j.pred$, $j.succ$, and $q.pred$ as long as j holds the locks.*

Proof. *We refer to j's successor ($j.succ$) as q. We refer to j's predecessor as p, which is $q.pred$ if j is about to join, and $j.pred$ if j is about to leave. Assume on the contrary that j's predecessor p is leaving. That would imply that p has acquired the locks L_p and $L_{p.succ}$, where $p.succ$ is either j or q depending on whether j is leaving or joining. Either way, it contradicts the fact that node j holds L_j and L_q. Similarly, assume that q is leaving the system, then q must have acquired the locks L_q and $L_{q.succ}$, contradicting that j holds the lock L_q. For the remaining part of the proof, there are two ways in which the pointers $p.succ$ and $q.pred$ can be altered. Either a node with j as successor tries to join, or a node with q as successor tries to join. Both cases are impossible as the locks L_j and L_q are held by the node j, and hence cannot be acquired by any other node. Node j's succ and pred pointers can be altered if j gets a new predecessor or a new successor. Both cases are impossible as a new predecessor would have to acquire L_j, and a new successor would have to acquire $L_{j.succ}$, both of which are already held by j.* □

The above theorem assumes that the system size is at least two. If a node is joining, the above theorem would even hold if the system size was 1. That would imply that the joining node j has acquired its own lock, L_j, as well as the lock of the remaining node q in the system. The theorem would be trivially true for that case as there are no other nodes that can interfere with the join operation, and q would not be able to leave as L_q would be held by node j while it is joining.

If the system size is 2 and j is leaving, j's successor and predecessor are the same node. The theorem will still hold, as j will acquire its own lock,

as well as its successors, and then complete its leave operation without any interference from any other node.

In summary, two special cases remain. If the system size is 1 and the single node wants to leave, it will not be able to acquire locks from itself and its successor, which is itself. Hence, we will detect that situation and let the node gracefully leave without any synchronization. We also need a similar special case if the joining node is the only node in the system.

The Dining Philosophers We have shown that a node will have mutually exclusive access to the pointers it needs by acquiring only two locks. By reducing the number of locks necessary by a joining/leaving node to two, certain aspects of our problem become reducible to the famous problem of the dining philosophers.

We briefly explain the original dining philosophers' problem. Five philosophers sit around a round table, on top of which a bowl of spaghetti is situated. There is a fork between any two neighboring philosophers, making it a total of five forks. Philosophers spend their time thinking and eating. To eat, a philosopher must be able to grab both her left and right fork. It thereafter leaves both forks and thinks for a while before repeating this pattern.

Our problem is similar to the problem just explained. The forks represent the locks. A joining or a leaving node represents a philosopher that wants to eat. By reducing our problem to the dining philosophers' problem, we can use many of the solutions previously provided to this problem. For example, the philosophers might end up in a *deadlock*, where each has picked up its right fork, and is waiting for its left fork. But since each fork is held by a different philosopher, we have a deadlock in which none of the philosophers can proceed to acquire a second fork.

We now define more precisely what we mean by a deadlock, so that we later can refine our algorithm to avoid deadlocks. Showing that an algorithm is deadlock-free is a safety property. Informally, a deadlock is associated with a "frozen" state where nothing whatsoever is being computed. In our case, we say that an algorithm can deadlock if an execution of the algorithm can reach a state in which one or more nodes are attempting to join or leave by acquiring the relevant locks, but they are each in standstill waiting for some lock that will never be released. Hence our aim is to show the safety property that a deadlock can never

occur.

Coffman *et al.* identified four *necessary conditions* for a deadlock to occur [29]:

1. *Mutual Exclusion*. The nodes claim exclusive control of the resources they require.

2. *Wait for*. Tasks hold resources already allocated to them while waiting for additional resources.

3. *No preemption*. Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion.

4. *Cyclic Wait*. A cyclic chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain.

Recall that if property X is a necessary condition for Y to happen, then if Y happens then property X must be true. Hence, the four necessary conditions for a deadlock imply that if a deadlock occurs, all four conditions must be true. Therefore, if one can show that any of the conditions never occurs for some algorithm, one has proved that a deadlock will never occur.

The tasks are in our case the nodes, and the resources that they want to access are the locks (or alternatively the pointers guarded by the locks). If we can show that any of them will never be true, there cannot be a deadlock. The mutual exclusion property is always true about the locks, as is the wait-for condition, as a node which has acquired one lock will be waiting to acquire the second. We can, however, design our algorithm to ensure that the remaining conditions are never true.

Without careful consideration, the cyclic wait condition can occur if, for instance, all nodes in the system want to leave. They would each acquire their own lock, and then wait to acquire the lock of their successor, which would never be released. If preemption is not possible, the system would deadlock.

One known solution to the dining philosopher's problem is to introduce asymmetry. We propose such a solution to avoid cyclic wait, which we call *asymmetric locking*. Let z be the node with the highest identifier. A node k can locally determine if it has the highest identifier if $k > k.succ$. If

node z attempts to leave the system, it should first attempt to acquire its successor's lock $L_{z.succ}$, and thereafter its own lock L_z. In any other case, where some node j wants to join or leave, it will first acquire its own lock L_j, and then thereafter acquire its successor's lock $L_{j.succ}$.

So far we have assumed that the pointers in the system are correct and that a node indeed manages to acquire its own lock and current successor's. This need not be the case. If a node ever tries to acquire a lock that is not free, the node will wait until it becomes free and then acquires it. The node which is waiting for a lock L_i will be notified by node i when the lock is free. This requires that node i queues requests to the lock it hosts in a *lock queue*, and notifies and removes one node in the queue each time L_i is released. Two additional operations are needed to ensure that nodes can properly acquire their successor's lock.

A leaving node's lock queue should be transferred to its successor. We first describe a naïve algorithm to achieve this, and later refine it. When a leaving node i has acquired all the relevant locks, it transfers its lock queue to its successor j, which will enqueue the lock queue of i onto its current lock queue. Hence, the elements in the lock queue of j maintain the same position in the queue after i leaves, while an element at position k in the lock queue of i gets position $k + l$, where l is the number of elements in j's lock queue before the merger of the lock queues. Hence, if some node i is waiting for its successor's lock $L_{i.succ}$ to become free, it will be notified even if its successors leave the system.

A joining node might need to take over parts of its successor's lock queue. When a joining node i has acquired all the relevant locks, its successor $i.succ$ transfers its lock queue to i. Node i will then remove from its lock queue every node that has $i.succ$ as its successor. Similarly, node $i.succ$ will remove from its lock queue every node that has i as its successor. More precisely, only nodes in the range $(i.succ, i]$ from $i.succ$'s lock queue are stored in i's lock queue, while only nodes in the range $(i, i.succ]$ from $i.succ$'s lock queue are stored in $i.succ$'s lock queue. Hence, if a node p is waiting for its successor's lock $L_{p.succ}$ and meanwhile gets a new successor q, it will be notified by the new successor q when the lock becomes free.

The above explained scheme will ensure that there will never be a cyclic wait.

Theorem 3.2.2. *The join and leave algorithms with asymmetric locking will*

never deadlock.

Proof. *We will show that the cyclic wait condition will never occur. Assume by contradiction that such a cyclic wait exists. A joining node's lock cannot be part of the cycle. The reason for this is that if a lock is part of a cyclic wait, some node must hold that lock and some other node must be waiting to acquire the same lock. However, a joining node j will only be part of the system after it finishes joining, and only thereafter some other node might attempt to acquire its lock L_j and form a cyclic wait.*

Since a leaving node will only attempt to acquire its own lock and its successor's, the cycle must follow the successor pointers. Hence, a cycle implies that there does not exist any node with a free lock.

Any cyclic wait consists of a cycle of at least two nodes, one of which is the node z with the highest identifier. Node z's lock is either held by z's predecessor y, or by z itself. If y holds L_z, node y must have already acquired L_y by the sequence of asymmetric locking, which implies that y has all the locks it needs. On the other hand, if L_z is held by node z, then z must have already acquired $L_{z.succ}$ by the sequence of asymmetric locking, in which case z has all the locks it needs. In either case, one of the nodes has all the locks it needs to proceed, and cannot be part of a cyclic wait, which contradicts that all nodes are part of the cyclic wait. □

3.2.2 Liveness

For liveness, there are several desirable properties. One desirable property is that the algorithm is free of *livelocks*. Informally, a livelock is a state in which none of the nodes can make progress toward their goal, while still taking steps. We say that an algorithm can livelock if it is possible that in an execution of the algorithm none of the nodes that want to join or leave can acquire the relevant locks. Freedom from livelock is a stronger requirement than freedom from deadlock. For example, every node in the system might attempt to leave by successfully acquiring its first lock. Thereafter, every node unsuccessfully attempts to acquire its successor's lock. After noticing that the successor's lock is not free, it might release its own lock and restart the whole procedure. If the nodes keep repeating this pattern forever, there is a livelock, but no deadlock.

We will not show that our algorithm is free of livelocks, as locks are never released prematurely as in the above scenario. Hence, livelocks

never occur.

A liveness property that is desirable for our algorithm is that it is free from *starvation*. Informally, starvation is when some node cannot make any progress. An algorithm suffers from starvation if it is possible in an execution of the algorithm that some node wants to join or leave, but is never able to acquire the relevant locks. Freedom from starvation is stronger than livelock-freedom, as some node might always be making progress, and hence be livelock free, even though a single node is not making any progress ever. In essence, livelock freedom ensures that some node is always doing progress, while starvation freedom ensures that every node makes progress.

There exist many solutions to the dining philosophers' problem that are starvation free. However, our problem is slightly different from the problem of the dining philosophers, as nodes are joining and leaving. Hence, the number of locks and philosophers is constantly changing. The joining and leaving of nodes can make nodes starve, as we show next.

The problem with the current algorithm is that when nodes leave, their lock queue is merged with their successor's lock queue. If some node is leaving, and its successor's lock queue is non-empty, the nodes in the lock queue of the leaving node will have a worse position after the lock queue of the leaving node is merged with the successor's lock queue. It is therefore conceivable that under conditions of continuous leaves and joins, some node j attempts to acquire a lock and ends up in a lock queue, which gets merged over and over with the successor's lock queue, resulting in node j never acquiring the desired lock.

We will therefore slightly modify our algorithm to ensure starvation freedom. We modify asymmetric locking to ensure that whenever a node attempts to acquire its own lock to leave, no other requests can be enqueued in its lock queue. This is realized by a *forwarding mechanism* as follows. As soon as a leaving node i attempts to acquire its own lock L_i, it will ensure that all further requests to its lock L_i are forwarded to its successor $i.succ$. This forwarding of requests makes sense as a leaving node i's request to acquire L_i indicates that i is about to leave, and requests enqueued after such a request should anyway be handled by i's successor after i has left the system.

We have now arrived at the full algorithm for asymmetric locking, as can be seen by Algorithms 2 and 3. Algorithm 2 mainly uses RPC notation, while the parts related to the forwarding mechanism (Algorithm 3)

use event notation. The reason for the use of event notation is that it simplifies describing the forwarding mechanism.

The algorithm uses the variable *LockQueue*, which represents a FIFO queue. The $Enqueue(m)$ procedure enqueues a request by node m in the lock queue. The $Dequeue()$ procedure simply removes the first element from the lock queue.

We now prove that asymmetric locking with the forwarding mechanism is starvation free. For that we need to introduce some simple notation.

Recall that a leaving node has to acquire its own lock and its successor's lock. Similarly, a joining node has to acquire its own lock and its successor's lock. Therefore, a lock queue can contain four types of requests: a request by a leaving node i to acquire its own lock L_i, a request by a leaving node i to acquire the lock of its successor, a request by a joining node i to acquire its own lock L_i, and a request by a joining node i to acquire the lock of its successor.

The lock queue and the four types of requests appearing in it are modeled as follows. The lock queue of a node i is represented by a sequence subscripted by the node identifier. The sequence $\langle\rangle_i$ represents an empty lock queue at node i, which indicates that lock L_i is free. The elements of the sequence are one of the symbols $\{\texttt{j}, \texttt{js}, \texttt{l}, \texttt{ls}\}$. The left-most element in the sequence is the first element in the lock queue, which represents the request currently holding the lock. The right-most element is the last element in the lock queue.

The symbols have the following meaning:

- The symbol `j` indicates a request by a joining node to acquire its own lock.
- The symbol `js` indicates a request by a joining node to acquire its successor's lock.
- The symbol `l` indicates a request by a leaving node to acquire its own lock.
- The symbol `ls` indicates a request by a leaving node to acquire its successor's lock.

For example, the sequence $\langle \texttt{js}, \texttt{js}, \texttt{ls}, \texttt{l} \rangle_5$ represents the lock queue at node 5. The first two items (`js`'s) in the lock queue represent requests by

Algorithm 2 Asymmetric locking with forwarding

```
1:  procedure n.Join(succ)                    ▷ Join the ring with succ as successor
2:      Leaving :=false                                          ▷ Initialize variable
3:      LockQueue.Enqueue(n)                  ▷ Enqueue request to local lock
4:      slock :=GetSuccLock()
5:      pred := succ.pred
6:      pred.succ := n
7:      succ.pred := n
8:      LockQueue := succ.LockQueue           ▷ Copy successor's queue
9:      LockQueue.Filter((pred, n])           ▷ Keep requests in the range
10:     succ.LockQueue.Filter((n, pred])      ▷ Keep requests in the range
11:     LockQueue.Dequeue()                   ▷ Remove local request
12:     ReleaseLock(slock)
13: end procedure

14: procedure n.Leave()                                          ▷ Leave the ring
15:     if n > succ then                                         ▷ Asymmetric Locking
16:         slock :=GetSuccLock()
17:         Leaving :=  true                                     ▷ Enable forwarding
18:         LockQueue.Enqueue(n)              ▷ Enqueue request to local lock
19:     else
20:         Leaving :=  true                                     ▷ Enable forwarding
21:         LockQueue.Enqueue(n)              ▷ Enqueue request to local lock
22:         slock :=GetSuccLock()
23:     end if
24:     pred.succ := succ
25:     succ.pred := pred
26:     LockQueue.Dequeue()                                      ▷ Remove local requst
27:     ReleaseLock(slock)
28: end procedure

29: procedure n.GetSuccLock()
30:     sendto succ.AcqLock(n)
31:     receive LockGranted() from m
32:     return m                              ▷ Return identity of lock host
33: end procedure

34: procedure n.ReleaseLock(dest)
35:     sendto dest.FreeLock()
36: end procedure
```

Algorithm 3 Asymmetric locking with forwarding continued

```
1: event n.AcqLock(src) from m
2:     if leaving = true then
3:         sendto succ.AcqLock(src)
4:     else
5:         LockQueue.Enqueue(src)              ▷ Enqueue src's request last
6:     end if
7: end event

8: event when New top element m in LockQueue at n
9:     sendto m.LockGranted()
10: end event

11: event n.FreeLock() from m
12:     LockQueue.Dequeue()                     ▷ Remove top element
13: end event
```

some joining nodes to acquire their successor's lock L_5. The third item in the lock queue (`ls`) is a request by the predecessor of 5, which wants to acquire L_5 in order to leave. The last item in the lock queue (`l`) is a request by node 5 to acquire L_5 to leave the system.

With the four symbols we can represent the lock queue at any given node at any time. We shall prove that any element in the lock queue will eventually reach the front of the lock queue, and hence every request to acquire a lock will eventually be granted.

Lemma 3.2.3. *If the symbol* `l` *occurs in a sequence, it must be the last element.*

Proof. *Assume the symbol* `l` *occurs in the sequence of node i. The symbol* `l` *indicates that node i is attempting to leave, and has thus requested to acquire its own lock L_i. As shown by the* AcqLock *event in Algorithm 2 line 3, any further requests to the lock queue of node i will be redirected to the successor of node i, hence no other requests can be enqueued after enqueueing* `l` *in the sequence representing the lock queue of node i. Furthermore, there can only be one* `l` *in any sequence, as a node cannot request to leave while it already has a pending leave request. Therefore* `l` *must be the last element of the sequence.*

□

Theorem 3.2.4. *Asymmetric locking with forwarding (Algorithm 2 and 3) is starvation free.*

Proof. *Notice that a joining node can always trivially acquire its own lock, since its lock queue is empty. So if the symbol* `j` *occurs in the sequence of node i, it must be the only symbol in the sequence, since node i is not yet part of the system and i is yet unknown to other nodes. Furthermore, notice that any symbol in a sequence can only improve (move toward the top element) or maintain its position in the queue. It remains to show that any symbol in a sequence will always improve its position in the queue.*

We will show that any symbol occurring in any lock queue will eventually reach the top position in the queue. Assume some symbol $s \in \{$`js`,`l`,`ls`$\}$ *occurs in the sequence of some node n. If s is the top element of the sequence we are done; the s request currently holds the lock. Assume s is not the top element of the sequence. According to Lemma 3.2.3 the* `l` *symbol can only be in the last position of a sequence and hence the symbol* `l` *cannot occur on the left side of symbol s. Hence, only symbols* `js` *and* `ls` *can occur on the left of symbol s in any sequence, which implies that the symbol occurring in the top position is either* `js`*, or* `ls`*. We inspect three cases separately.*

Case 1; Assume n is the node with the highest identifier ($n = z$)*. Regardless of whether the top element is* `js`*, or* `ls`*, it represents a request by some node m to acquire the second and final lock. Hence, m has acquired both required locks and will soon release both of them by calling Dequeue*().

Case 2; Assume n is the successor of the node with the highest identifier ($n = z.succ$)*. If the top element is* `js`*, the node making the request has acquired both its locks and will eventually be dequeued from the sequence. If the top element is a* `ls`*, it represents a request made by node z. That implies that z has acquired its first lock, and z will request* L_z*, which by case 1 will eventually be granted, after which z has both required locks implying that* `ls` *will eventually be dequeued from the sequence.*

Case 3; Assume n is any other node other than z and z.succ. This case is the same as case 1.

All three cases show that the top element will repeatedly be dequeued, until the top element becomes s, which completes the proof that any request to a lock will eventually be granted.

□

Drawbacks with Asymmetric Locking There are some performance drawbacks with the proposed asymmetric locking scheme. If neighboring nodes on the ring all try to leave at the same time, it might in the worst case happen that they can only make progress sequentially, one-by-one. Assume a system consisting of 10 nodes with the identifiers 5, 6, $\cdots$, 14. As indicated by Figure 3.3, nodes 5, 6, 7, 8, 9, might all attempt to leave the the same time. Each of the nodes i successfully acquires its own lock L_i. Thereafter, nodes 5 through 8 attempt to take the lock hosted by their successor, but as the lock is currently held by the hosting node, their request is forwarded until it ends up in the lock queue of node 10. Only node 9 will succeed in acquiring L_{10}, and then successfully leave. Thereafter, node 8, which is now placed on node 10's lock queue, can acquire L_{10} and then leave. This continues sequentially in this manner, until finally node 5 acquires L_{10} and leaves the system. The above situation can be generalized to n neighboring nodes leaving, in which it will take time linearly proportional to n before all of them are done leaving. In addition, if any node wants to join, and its successor is one of the leaving nodes, the joining node has to wait as well.

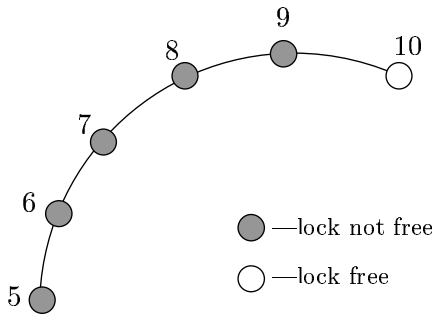

Figure 3.3: Consecutive leaves leading to sequential progress. Nodes 5 through 9 are attempting to leave, each has acquired its own lock, and is waiting for its successor's lock. Only node 9 can make progress by acquiring L_{10}, thereafter node 8 makes progress, etcetera.

To circumvent the above situation, we provide another solution which is inspired by the third Coffman condition: preemption of nodes that hold a lock. Since the join/leave algorithms only modify pointers after they have acquired two locks, a node which manages to get one lock, but fails to get a second lock, could release the first lock and retry.

Our *randomized locking* algorithm works as follows. Every joining/leaving node j first attempts to acquire its own lock L_j, and thereafter its successor's lock, $L_{j.succ}$. If a node cannot acquire some lock because the lock is not free, the node releases all the locks it holds and retries to acquire the locks again after waiting a random time.

Aside from the performance reasons previously mentioned, this solution is simpler as it is stateless, and hence simplifies fault-tolerance. For example, if some node fails, all the nodes in its lock queue will be waiting indefinitely for it.

We first state a simple fact, and then show that the algorithm is starvation free.

Theorem 3.2.5. *The randomized locking algorithm is free from deadlocks.*

Proof. *The third Coffman condition, preemption of locks, is never satisfied. Therefore, the necessary conditions for a deadlock are never satisfied.* □

Hence, the randomized locking algorithm ensures that every held lock is eventually released, either because a node has acquired both necessary locks and will release both locks after updating the relevant pointers, or because the node holding the lock was not able to acquire both necessary locks and will therefore release any acquired locks to try again later.

Next, we show that the algorithm is free from starvation assuming some finite bound on the total number of joins and leaves. The assumption is justified because there can only be a finite number of nodes that can contend for the lock at any given point in time.

Theorem 3.2.6. *The randomized locking algorithm is free from starvation.*

Proof. *Assume the maximum number of nodes that can contend for a lock at any given instant is k. Theorem 3.2.5 showed that the lock is always freed, in which case all the nodes race to acquire it. One of them will always succeed. We assume that all nodes contending for a lock have equal probability of succeeding. This is motivated by the random wait in the algorithm.*

The probability that a fixed node j is never able to fetch its first lock and starve is:

$$\Pr[j \text{ starves}] = \lim_{n\to\infty} \left(1 - \frac{1}{k_1}\right)\left(1 - \frac{1}{k_2}\right)\cdots\left(1 - \frac{1}{k_n}\right)$$

Where k_i is the number of contending nodes time i, hence $k_i \leq k$. Therefore,

$$\Pr[j \text{ starves}] \leq \lim_{n \to \infty} \left(1 - \frac{1}{k}\right)^n = 0$$

The above argument shows that every node will eventually get its first lock. The argument can be extended to the second lock as well, as a node that acquired its first lock will contend for the second lock. Even if it is not able to get the second lock, it will be able to eventually acquire its first lock, and again contend for its second lock. Hence, a node keeps contending for its second lock, and will eventually acquire it by the same argument as above.

□

3.3 Lookup Consistency

The previous section primarily dealt with concurrency control. It showed how concurrent join and leaves could be coordinated to avoid two neighboring nodes in the ring joining and/or leaving at the same time. So far, we have not dealt with the traversal of these pointers, i.e. lookups. While joins and leaves are happening, we would like to make lookups to find the successor of certain identifier.

Correct lookups in the presence of dynamism is not only important for applications using the overlay, it is crucial to make joins work properly. In the algorithms described in the previous section we assumed that a joining node knows its successor. For this assumption to be valid, a joining node needs to acquire a reference to its successor, which it does by making a lookup.

Correctness of lookups will depend on the lookup algorithm, as well as the join and the leave algorithms. So far, we have only explained how a node successfully acquires locks to avoid conflicting updates to pointers. We have to, however, ensure that potential lookups are correct when the *succ* and *pred* pointers are being updated during join and leave operations. Next, we show how a joining or leaving node should update the relevant pointers when it has acquired the necessary locks. Since we assume the relevant locks are acquired, the *succ* and *pred* pointers can be updated without the interference of any other joins or leaves (see the Non-interference Theorem (3.2.1)).

3.3.1 Lookup Consistency in the Presence of Joins

In Section 3.2 we showed how a node acquires the relevant locks. In this section we describe how a joining node, which has acquired both relevant locks, updates its own, as well as its successor's and predecessor's *succ* and *pred* pointers. We refer to the joining node as q, its predecessor as p, and its successor as r.

Algorithm 4 assumes that some joining node has acquired both relevant locks, and therefore has a correct *succ* pointer. We also assume that its *pred* pointer is set to `nil`. The time-space diagram shown by Figure 3.4 depicts the same algorithm fully. Time-space diagrams normally only show one out of many possible executions. However, Algorithm 4 has no alternative executions, or interleavings and therefore the time-space diagram contains all information about the algorithm.

As seen by Figure 3.4, the joining node q sends an UpdatePred message to its successor r. The successor r, upon receipt of UpdatePred, sets a special boolean variable called JoinForward to `true`, updates its *pred* pointer to point to the joining node q, and sends a JoinPoint message to the joining node. The receipt of the UpdatePred message constitutes a *join point*, which represents that responsibility of the identifiers in the range $(p, q]$ are instantaneously transferred from r to q. The rest of the algorithm is straight forward, as the joining node updates both its pointers, sends an UpdateSucc message to its predecessor p, which then sends a StopForwarding message to its successor r and updates its successor pointer to point to the newly joined node. Node r sets its special `JoinForward` variable to `false` upon receipt of StopForwarding, and terminates the algorithm by sending Finish to the joining node. The joining node knows the pointers have been updated correctly when it receives Finish, and can safely release any held locks.

Any node in the system might do a lookup while nodes are joining. During a join, however, node p's successor pointer might point to either node r or node q. We would like it to point to r before the join point, and to q after the join point. The former case is ensured automatically assuming p's successor pointer was correctly pointing to r before the join operation. The latter case, however, is not necessarily satisfied. We however circumvent the problem by letting r forward requests coming from p ($r.oldpred$) to node q while r's variable *JoinForward* is true. The FIFO requirement for channels ensures that messages from p pass through node

Algorithm 4 Pointer updates during joins

```
1:  event n.UpdateJoin() from n                    ▷ Assuming succ is correct
2:      sendto succ.UpdatePred()
3:  end event

4:  event n.UpdatePred() from m
5:      JoinForward :=true                          ▷ Forwarding Enabled
6:      sendto m.JoinPoint(pred)                    ▷ Join Point
7:      oldpred := pred
8:      pred := m
9:  end event

10: event n.JoinPoint(p) from m
11:     pred := p
12:     succ := m
13:     sendto pred.UpdateSucc()
14: end event

15: event n.UpdateSucc() from m
16:     sendto succ.StopForwarding()
17:     succ := m
18: end event

19: event n.StopForwarding() from m
20:     JoinForward :=false                         ▷ Forwarding Disabled
21:     sendto pred.Finish()
22: end event
```

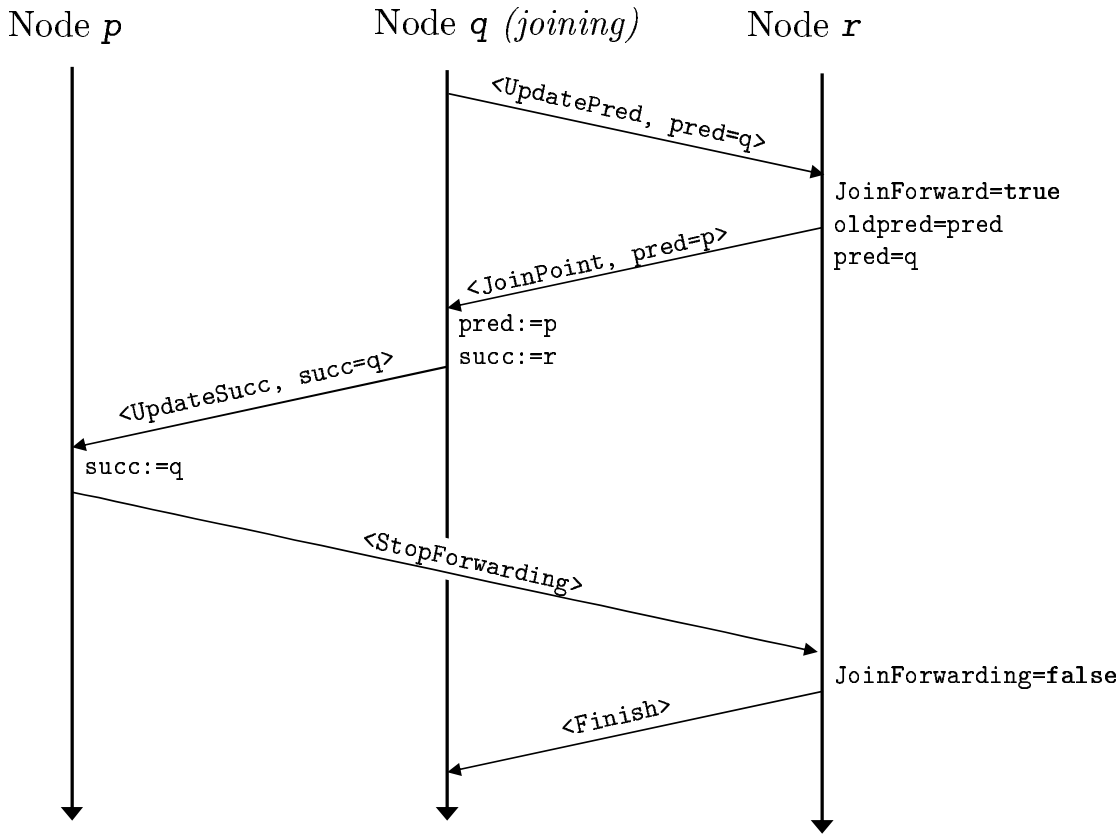

Figure 3.4: Time-space diagram showing how a joining node should update the relevant *succ* and *pred* pointers. Node q should have acquired the relevant locks before initiating the algorithm, and it should release the locks when the algorithm finishes.

q after the join point.

3.3.2 Lookup Consistency in the Presence of Leaves

In this section we describe how a leaving node, which has acquired both relevant locks, updates its successor's and predecessor's *pred* and *succ* pointers, respectively. We refer to the leaving node as q, its predecessor as p, and its successor as r.

Algorithm 5 assumes that some leaving node has acquired both relevant locks. The time-space diagram shown by Figure 3.5 depicts the same algorithm fully.

As seen by Figure 3.5, the leaving node q starts by setting its boolean *LeaveForward* variable to `true` and sends a LEAVEPOINT message to its successor r. This constitutes a *leave point*, which represents that responsibility of the identifiers in the range $(p, q]$ are instantaneously transferred from q to r. The rest of the algorithm is straightforward, as node r updates its predecessor pointer to point to p and informs p to update its successor pointer to point to r. Thereafter, node p sends a STOPFORWARDING message to q. Node q sets its special *LeaveForward* variable to `false` upon receipt of STOPFORWARDING.

The leaving node knows the pointers have been updated correctly when it receives STOPFORWARDING, and can safely release any held locks and leave the system.

Algorithm 5 Pointer updates during leaves

```
1:  event n.UPDATELEAVE() from n
2:      LeaveForward := true                    ▷ Forwarding Enabled
3:      sendto succ.LEAVEPOINT(pred)
4:  end event

5:  event n.LEAVEPOINT(p) from m
6:      pred := p
7:      sendto pred.UPDATESUCC()
8:  end event

9:  event n.UPDATESUCC() from m
10:     sendto succ.STOPFORWARDING()
11:     succ := m
12: end event

13: event n.STOPFORWARDING() from m
14:     LeaveForward :=false                    ▷ Forwarding Disabled
15: end event
```

As with the join case, any node in the system might do a lookup while nodes are leaving. During a leave, however, node p's successor pointer might point to either node r or node q. We would like it to point to q before the leave point, and to r after the leave point. The former case is ensured automatically assuming p's successor pointer was correctly

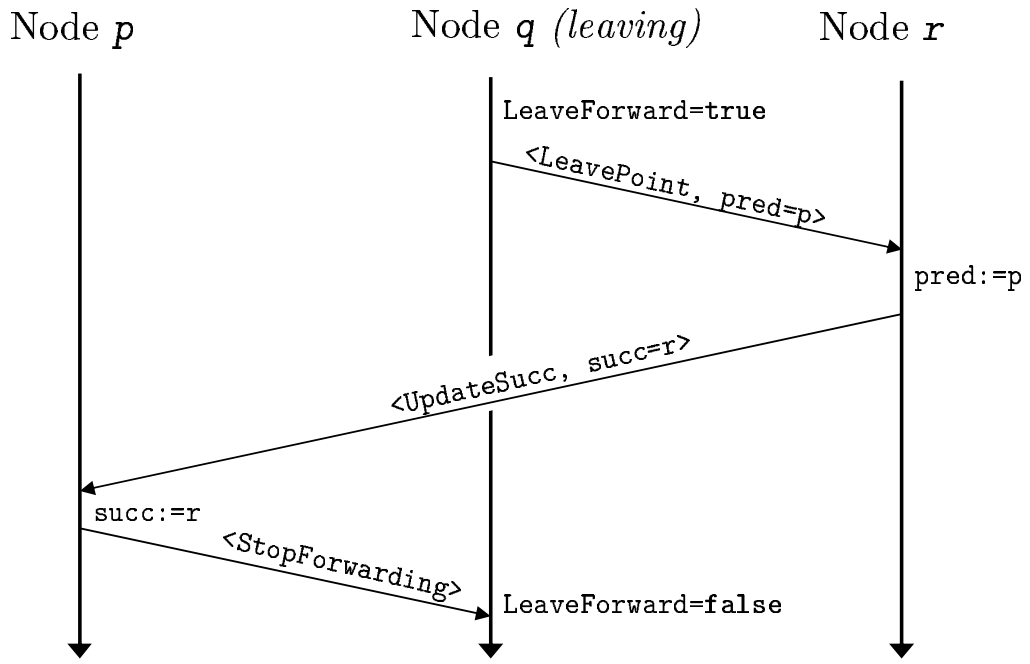

Figure 3.5: Time-space diagram showing how a leaving node should update the *succ* and *pred* pointers. Node *q* should have acquired the relevant locks before initiating the algorithm, and it should release the locks when the algorithm finishes.

pointing to *r* before the leave operation. The latter case, however, is not necessarily satisfied. We however circumvent the problem by letting *q* forward requests coming from *p* to node *r* while *q*'s variable *LeaveForward* is true. The FIFO requirement for channels ensures that messages from *p* pass through node *r* after the leave point.

3.3.3 Data Management in Distributed Hash Tables

So far, we have only mentioned that identifier responsibility moves from one node to another as nodes join and leave. As we previously mentioned, the concept of identifier responsibility can be used to build a distributed hash table (DHT) abstraction. In such a case, a node might be locally storing data items, whose keys are in the range of the node's identifier responsibility. As identifier responsibility changes, so do the items that a node should be storing.

We first present naïve solution. As a node's responsibility is changed by the sending of a JoinPoint or LeavePoint, items in the changed ranged can be piggy-backed with the message, ensuring that data items are always present at the right place.

As the size of the data items grow, it might be infeasible to piggy-back all necessary items in one message. Nevertheless, what is important is that data responsibility is always consistently defined, which we will show is the case with our algorithms. Another protocol could be used, which lazily, or eagerly fetches items according to the data responsibility. For example, as data responsibility shifts with the sending of a LeavePoint message, the successor of the leaving node could buffer all requests to the identifiers in the changed range, while the leaving node transfers the items over to its successor. Whenever the successor of the leaving node has received all items of the leaving node, it can begin to process the buffered queries. A similar scheme can be used for joins.

3.3.4 Lookups With Joins and Leaves

The previous sections paved the way for the lookup algorithm, which we now fully define.

Algorithm 6 shows a *transitive lookup*, which goes from node to node until it arrives at the successor of the identifier, in which case it returns directly to the source of the request. The algorithm is initiated by sending a Lookup(id, src) message to any node, where id is the identifier whose successor is to be found, and src is the source node to receive the response.

The algorithm first checks if the *JoinForward* variable is true, in which case it ensures that messages from its predecessor's predecessor (the *oldpred* variable) are redirected to its predecessor. A similar check is made if the variable *LeaveForward* is true, in which case the node knows it is leaving, and hence forwards the message to its successor. Note that *JoinForward* and *LeaveForward* cannot both be true, as that would indicate that the current node is leaving while its predecessor is joining, which contradicts the locking mechanism described in Section 3.2.

If both *JoinForward* and *LeaveForward* are false, the algorithm first checks to see if *pred* is `nil`. This can happen if a joining node initiates a lookup before reaching its join point, in which case it forwards the query to its successor. Otherwise, if the destination identifier is in its own responsibility, it responds with an answer. In any other case, it forwards

the message along the ring to its successor.

Algorithm 6 Lookup algorithm

```
1: event n.Lookup(id, src) from m
2:     if JoinForward = true and m = oldpred then
3:         sendto pred.Lookup(id, src)                ▷ Redirect Message
4:     else if LeaveForward = true then
5:         sendto succ.Lookup(id, src)                ▷ Redirect Message
6:     else if pred ≠ nil and id ∈ (pred, n] then
7:         sendto src.LookupDone(n)
8:     else
9:         sendto succ.Lookup(id, src)
10:    end if
11: end event
```

Proving Correctness of Lookup Consistency Our consistency requirement will be that at any given time, every identifier will be under the responsibility of exactly one node.

More formally, we say that the *configuration* of the system at any given discretized time, is the nodes in the system and their *succ*, *pred* pointers as well as their variables *JoinForward*, *LeaveForward*, and *oldpred*.

We now construct a function, which given a configuration, mimics the lookup operation of the system. For any given configuration of the system δ, we define a function called $lookup_\delta$ that takes two identifiers k and i, where k is some arbitrary destination identifier and i is the identifier of a node in δ, and returns the identifier of some node in δ. We do not provide the function, but it looks almost identical to Algorithm 6, except that the message passing is replaced with recursive calls.

Our consistency requirement can therefore be defined as:

$$\text{if } lookup_\delta(k,i) = p \text{ and } lookup_\delta(k,j) = q, \text{ then } p = q$$

The above requirement ensures that if the system state is frozen at any given instant, lookups for any identifier will return the same responsible node regardless of the node at which the lookup is initiated.

Theorem 3.3.1. *The lookup algorithm satisfies the consistency requirement.*

Proof. *We first proceed by induction on joins. The hypothesis is that the consistency requirement is true for a configuration.*

First, notice that the first node ever is handled as a special case, where the joining node j sets $j.succ = j$ and $j.pred = j$, making it responsible for all lookups. Hence, the hypothesis is trivially true for the base case.

Assume the hypothesis is true for some configuration δ. Then we show that it will be true for all configurations which result from the steps of the join algorithm. Assume node q is joining, with predecessor p and successor r. Before q joins, $r.pred$ is pointing to p, making $lookup_\delta(k, r) = r$ for all keys k in $(p, r]$, and by the hypothesis $lookup_\delta(k, i) = r$ for all nodes i in δ.

In the first step of q's join, $q.succ$ is set to r and $q.pred$ is set to `nil`*. This implies that lookups are unaffected, as any lookup from q will be forwarded to r, and lookups do not terminate at q since $q.pred$ is set to* `nil`*.*

The second step is the join point when r receives UpdatePred*, sets $r.pred$ to point to q, and enables join forwarding. From thereon, lookups for identifiers $(p, q]$ will return q regardless of where they are initiated. If initiated by r, they are forwarded to q since join forwarding is on. If initiated by q, they will be forwarded to r which redirects it to q, which by the FIFO assumption has set $q.pred$ to p, and hence will return itself as responsible. If they are initiated anywhere else, they will by the induction hypothesis end up at node r, which forwards them to node q, which returns itself as responsible. The next step, the receipt of* UpdateSucc *by p, does not affect the results of lookups, but merely incorporates q into the chain of successors. It remains to show that the step where r turns of join forwarding does not affect lookups. By the FIFO assumption, the receipt of* StopForwarding *ensures that $q.succ = r$, $q.pred = p$, $p.succ = q$, $r.pred = q$, i.e. q is properly incorporated into the ring, therefore forwarding is no longer necessary.*

The existence of configurations where the hypothesis is true due to join has been shown. We now show change our hypothesis to be that the consistency requirement is true for δ or δ contains no nodes. Assume the hypothesis is true for δ, we then show that if q (with predecessor p and successor r) leaves, it hypothesis will be true for all intermediary configurations. If q is the last node, then the hypothesis is trivially true. Otherwise, by the hypothesis, all lookups for $(p, q]$ terminate at q with q as responsible. In the first step, leave forwarding is enabled by q. Hence, any lookups terminating in δ at node q, will be forwarded to node r which will, by the FIFO assumption, have $r.pred = p$. Therefore, any queries previously returning q as responsible will return r as responsible. Second step makes $r.pred = p$, ensuring lookups to identifiers in $(p, q]$ reaching r are

terminated with r as responsible. Note that the second step causally succeeds the first step, ensuring that requests to q are forwarded to r. The third step ensures that p.succ = r, r.pred = p, and leave forwarding is enabled, hence there are no pointers to q in the configuration. Finally, q safely disables leave forwarding, as no more lookups could arrive to q as of the third step.

This completes the proof that the consistency requirement is always satisfied. □

3.4 Optimized Atomic Ring Maintenance

In this section we combine the randomized locking algorithm, and the lookup consistency algorithm, with all required special cases for system sizes less than three and describe the algorithms.

It is possible to combine the asymmetric or randomized locking scheme with the pointer update algorithm (Algorithms 4 and 5) to arrive at a full algorithm. The algorithm can, however, be optimized to consume less messages. This can be realized by a close look at the asymmetric locking algorithm (Algorithm 2). A joining or leaving node has to acquire its successor's lock, which requires two messages. Only thereafter it can update the successor's *pred* pointer, a step which also requires two messages. This section optimizes these two steps such that a successful request to acquire the successor's lock will have the side effect that the successor correctly updates its *pred* pointer.

General Algorithm Description The lock at each node is represented by the variable *lock*, which takes two possible values {`free`,`taken`}, initially set to `free`. Similarly, each node uses two boolean variables called *JoinForward* and *LeaveForward*, which are initially set to `false`.

Each node also keeps a variable called *status*, which is only used to facilitate the understanding of the algorithm. The *status* variable changes values according to the state machine shown in Figure 3.6. The state called `inside` indicates that the node is not leaving nor joining, nor is its predecessor leaving. The rest of the states are explained, below, in the informal descriptions of the algorithms.

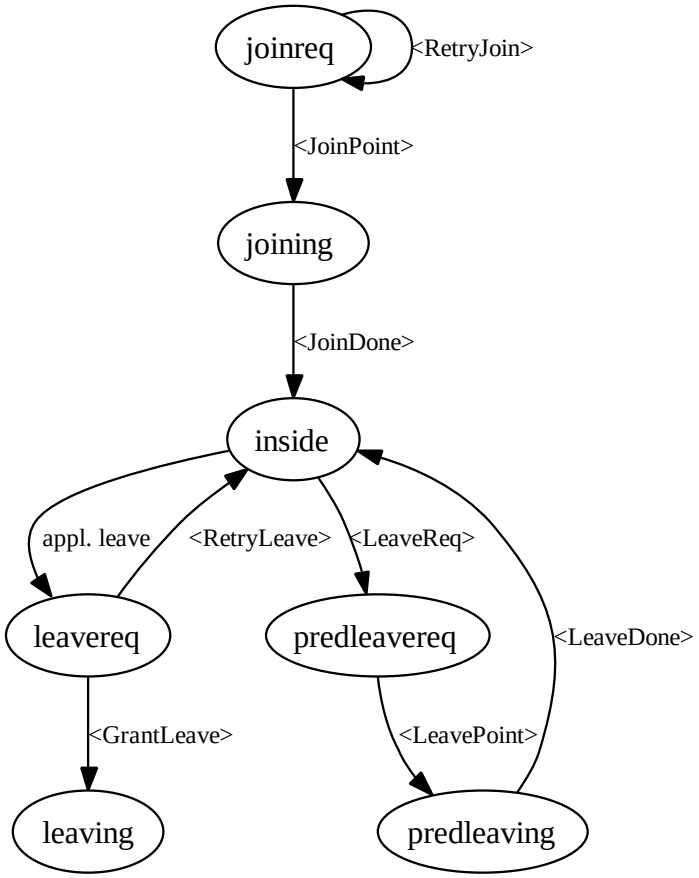

Figure 3.6: State transition diagram showing how a nodes status can change for the optimized randomized algorithm. Events indicate received messages, while the states indicate the status of the node.

3.4.1 The Join Algorithm

We now informally describe the join algorithm, which is given by Algorithms 7 and 8. Throughout the example, we will assume that a node q is joining between a node r and its predecessor p.

Initially, a joining node starts with *lock* set to `taken` and *status* set to `joinreq`, indicating that it has acquired the local lock and it is waiting to join. An exception is made if the node is the only node in the system, in which case it initializes its pointers, sets its lock to `free`, and sets *status* to the state `inside`. The next step for the joining node with id q is to send a JoinReq message to the current successor of identifier q. This is trivially done by following the successor pointers until a node r is found where q

is an identifier which is under the responsibility of r ($q \in (r.pred, r]$). We are currently not really concerned with the efficiency or the algorithmic details of finding q's successor, but we shall return to this issue later in Chapter 4.

The successor r of a joining node q will either grant q's request or asks q to retry joining later. The latter case occurs when r's lock is taken, in which case r sends q a RETRYJOIN message, which results in q waiting a random amount of time before retrying. This scheme can be optimized by letting the successor preempt the retry when its lock becomes free.

If node r grants q's join request, r will immediately set its boolean variable *JoinForward* to `true` and change the state of its lock to `taken`, indicating that it is locked because its predecessor is joining. It will also save its *pred* pointer in a temporary *oldpred* variable, and change its *pred* pointer to point to the joining node q. Thereafter r will send q a JOINPOINT message, which constitutes the *join point*, where the identifiers in the range $(r.oldpred, q]$ are instantaneously transferred to the new node q. Node q updates its successor and predecessor variable whenever it receives the JOINPOINT from its successor, and updates its status variable from `joinreq` to `joining`, indicating that the join point has occurred. Hence, both the nodes involved in the move of the join point can determine from their variables if their join point has occurred.

Finally, after receiving the JOINPOINT message, the new node q will ask the predecessor to update its *succ* pointer. This is achieved by sending a NEWSUCC message to the predecessor, which responds by updating its *succ* variable to q and sends a NEWSUCCACK to its old successor r ($p.succ$), which will free its lock and set its status to `inside`. Thereafter, r sends a JOINDONE message to the new node, which finally frees its lock.

As previously described, a node with $JoinForward = \texttt{true}$ will redirect messages received from *oldpred* to the new node (*pred*) to ensure that lookups relevant to the new node always end up at the new node after the join point. Hence, lookup consistency is always guaranteed (see lookup consistency in Section 3.3.4).

A successful execution of a join operation is shown by the time-space diagram shown in Figure 3.7.

Algorithm 7 Optimized atomic join algorithm

```
1:  event n.Join(e) from app
2:      if e = nil then
3:          lock := free
4:          pred := n
5:          succ := n
6:      else
7:          lock := taken
8:          pred := nil
9:          succ := nil
10:         status := joinreq
11:         sendto e.JoinReq(n)
12:     end if
13: end event

14: event n.JoinReq(d) from m
15:     if JoinForward and m = oldpred then
16:         sendto pred.JoinReq(d)                    ▷ Join Forwarding
17:     else if LeaveForward then
18:         sendto succ.JoinReq(d)                    ▷ Leave Forwarding
19:     else if pred ≠ nil and pred ≠ n and d ∈ (n, pred] then
20:         sendto succ.JoinReq(d)
21:     else
22:         if lock ≠ free or pred = nil then
23:             sendto m.RetryJoin()
24:         else
25:             JoinForward := true
26:             lock := taken
27:             sendto d.JoinPoint(pred)
28:             oldpred := pred
29:             pred := d
30:         end if
31:     end if
32: end event
```

Algorithm 8 Optimized atomic join algorithm continued

```
1: event n.JoinPoint(p) from m
2:     status :=joining
3:     pred := p
4:     succ := m
5:     sendto pred.NewSucc()
6: end event

7: event n.NewSucc() from m
8:     sendto succ.NewSuccAck(m)
9:     succ := m
10: end event

11: event n.NewSuccAck(q) from m
12:     lock := free
13:     JoinForward := false
14:     sendto q.JoinDone()
15: end event

16: event n.JoinDone() from m
17:     lock := free
18:     status := inside
19: end event
```

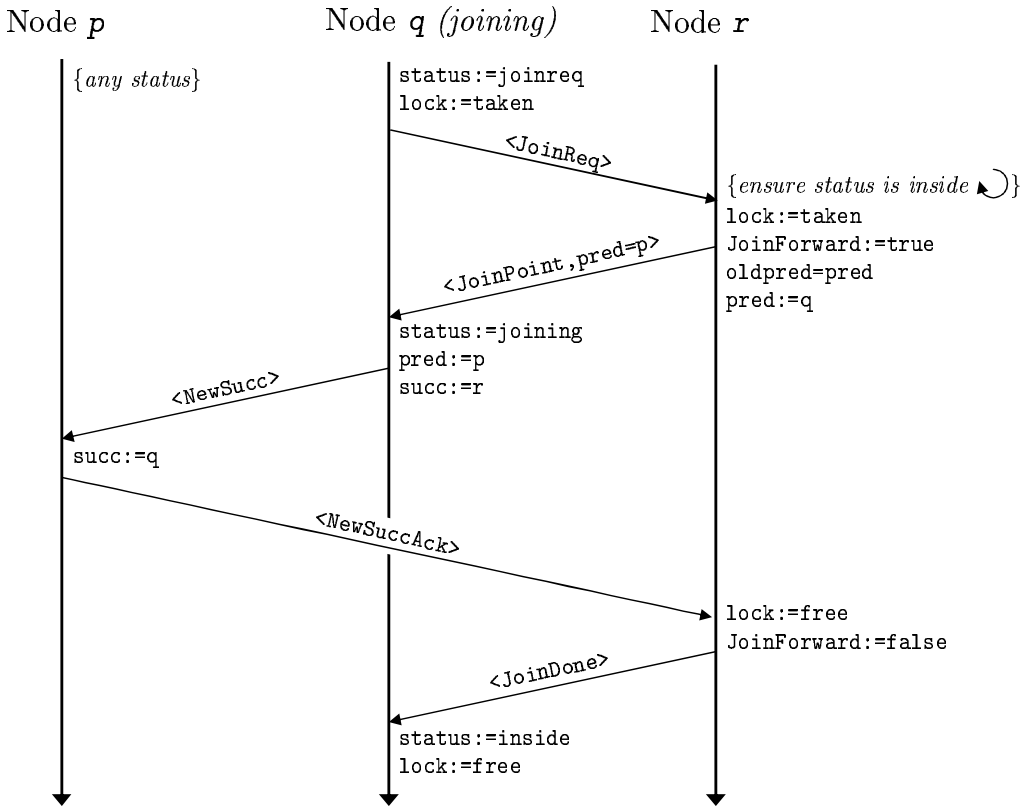

Figure 3.7: Time-space diagram of the successful join of a node.

3.4.2 The Leave Algorithm

We now informally describe the leave algorithm, which is given by Algorithms 9 and 10. Throughout the example, we will assume that a node q is leaving with predecessor p and successor r.

The leaving node q can only initiate a leave request when its lock is free. If it is not, it will wait and retry later. When its lock is free, it initiates the leave operation. If the node is the last node in the system, it will detect that, since its its *pred* and *succ* pointers will be pointing at itself, in which case it can leave unnoticed. If it is not the last node, it starts by sending a LeaveReq to its successor r.

The successor, node r, will only accept a leave request if its lock is free. If it is not, it will send a RetryLeave message, which results in q freeing its lock and waiting a random amount of time before retrying again. If r accepts the request, it sets its lock to `taken` and it changes its status from

`inside` to `predleavereq` and sends a GRANTLEAVE message to the leaving node q.

Upon receiving the GRANTLEAVE message, the leaving node sets its variable *LeaveForward* to `true`, changes its status to `leaving`, and transfers responsibility of all identifiers in $(q.pred, q]$ to its successor r. We will call this the *leave point*. This is done by sending a LEAVEPOINT message to the successor r, which reacts by changing its status from `predleavereq` to `predleaving` and setting its *pred* pointer to the leaving node's predecessor, p.

After the leave point, r asks its new predecessor to update its *succ* pointer to point to r by sending a UPDATESUCC message to p. Node p, reacts by sending UPDATESUCCACK to its current successor q, and thereafter updating its *succ* pointer to point to r. The leaving node q knows by the receipt of UPDATESUCCACK that its predecessor its no longer going to forward any queries to it, and can therefore send a LEAVEDONE message to its successor r and leave the system.

Finally, node r receives LEAVEDONE, frees its lock, and changes its status to `inside`, to allow new join or leaves, either from itself, its predecessor, or from new nodes.

As with joins, misdirected messages are redirected. In particular, any messages received will be redirected to the successor of the leaving node to ensure lookup consistency (see lookup consistency in Section 3.3.4).

A successful execution of a leave operation is shown by the time-space diagram shown in Figure 3.8.

3.5 Dealing With Failures

Our purpose is to build a system which functions in an asynchronous network, such as the Internet. It is therefore natural to aim at providing lookup consistency in the presence of crash failures and network partitions.

Unfortunately, we will show that it is impossible to implement a system which provides lookup consistency in an asynchronous network with network partitions. The result is related to what is known as *Brewer's Conjecture* [19], which states that it is impossible for a web service to provide the following three guarantees:

- Consistency

Algorithm 9 Optimized atomic leave algorithm

```
1: event n.Leave() from app
2:     if lock ≠ free then                    ▷ Application should try again later
3:     else if succ = pred and succ = n then
                                                ▷ Last node, can quit
4:     else
5:         status := leavereq
6:         lock := true
7:         sendto succ.LeaveReq()
8:     end if
9: end event

10: event n.LeaveReq() from m
11:     if lock = free then
12:         lock := taken
13:         sendto m.GrantLeave()
14:         state :=predleavereq
15:     else if lock ≠ free then
16:         sendto m.RetryLeave()
17:     end if
18: end event

19: event n.RetryLeave() from m
20:     status := inside
21:     lock := free                           ▷ Retry leaving later
22: end event

23: event n.GrantLeave() from m
24:     LeaveForward := true
25:     status := leaving
26:     sendto m.LeavePoint(pred)
27: end event
```

Algorithm 10 Optimized atomic leave algorithm continued

```
1: event n.LeavePoint(q) from m
2:     status :=predleaving
3:     pred := q
4:     sendto pred.UpdateSucc()
5: end event

6: event n.UpdateSucc() from m
7:     sendto succ.UpdateSuccAck()
8:     succ := m
9: end event

10: event n.UpdateSuccAck() from m
11:     sendto succ.LeaveDone()                    ▷ Leave the system
12: end event

13: event n.LeaveDone() from m
14:     lock := free
15:     status := inside
16: end event
```

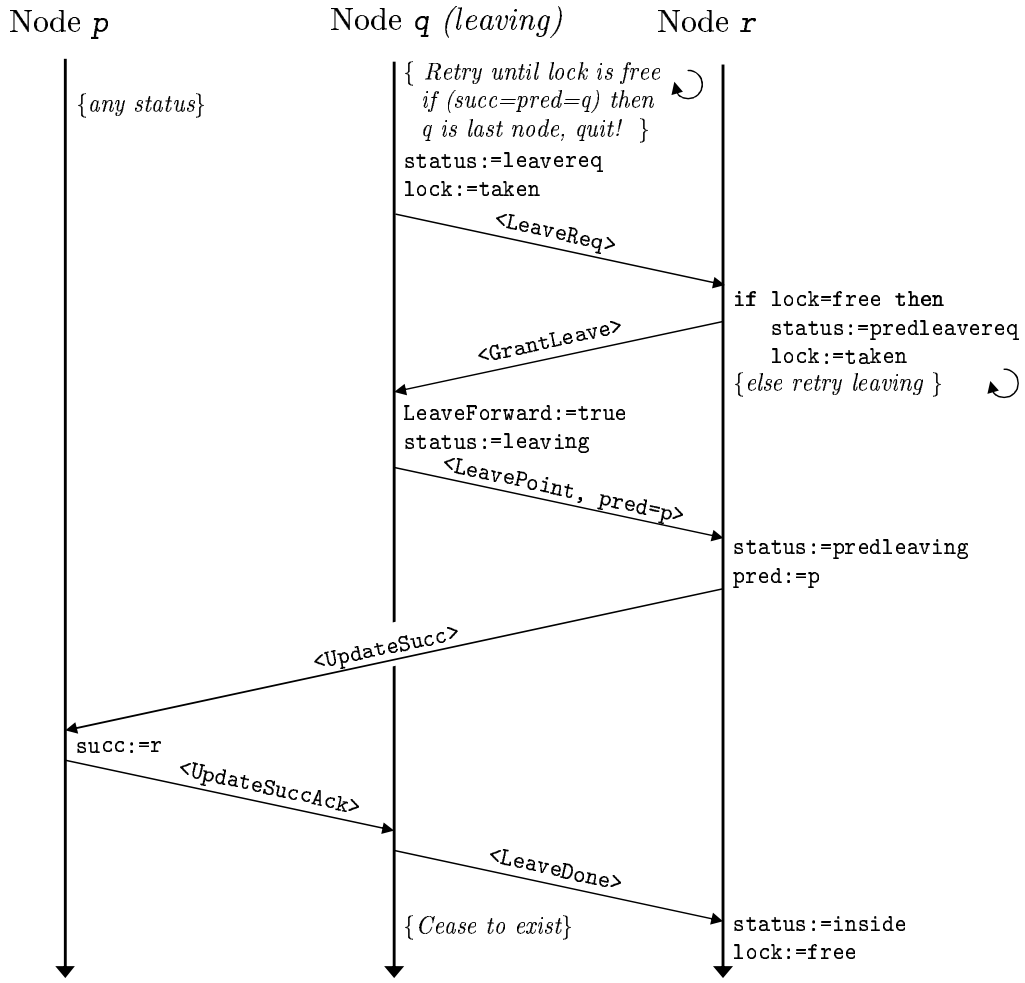

Figure 3.8: Time-space diagram of the successful leave of a node.

- Availability
- Partition-tolerance

The conjecture has been formalized and proven by Gilbert and Lynch [52]. We will take consistency to be lookup consistency as we defined in Section 3.3.4. We next describe the term availability and partition-tolerance.

By availability, it is meant that every request received by a non-failed node must eventually result in a response. This requirement is quite weak, as it does not require a response within any time bounds, but

rather requires that a response comes back at some point in time. Hence, it is a natural termination requirement for any distributed service.

Partition tolerance[1], means that the nodes in the system can become partitioned into different components, in which nodes in different components cannot communicate.

We now give the impossibility result, which even allows for inconsistent lookups while the network is partitioned. The proof makes certain assumptions about lookups, because it is trivial to create a system which guarantees lookup consistency by always returning 0 as the result of any lookup. More precisely, we assume that a lookup returns the identity of one of the nodes that is in the same partition as the initiator of the lookup, and that the identity of all nodes is unique. The Chord lookup function, which returns the successor of the identifier satisfies this requirement, given that the responsible node is in the same network component as the lookup initiator.

Theorem 3.5.1. *It is impossible in the asynchronous network model to provide a ring-based structured overlay network that guarantees the following properties:*

- *Lookup consistency in every network component*
- *Availability*
- *Partition tolerance*

Proof. *The proof proceeds by contradiction. Assume there exists a system which guarantees availability, partition tolerance, and provides lookup consistency in every network component.*

Assume a configuration C of a correct ring consisting of the nodes 0, 1, 2, 3, 4, 5. *Assume the network partitions the nodes into the following two components* $A = \{2, 3, 4\}$ *and* $B = \{0, 1, 5\}$.

The system still needs to provide availability. Hence, a lookup for identifier x in component A needs to return an identifier $i \in A$. *Therefore, some operations* O_A *will take place on the nodes in A which adapt the pointers in A, such that the lookup returns an identifier* $i \in A$. *Similarly, a lookup for identifier x in component B needs to return an identifier* $i \in B$. *Therefore, some operations* O_B *will take place on the nodes in B which adapt the pointers in B, such that*

[1]Gilbert and Lynch model a partition as a network which is allowed to lose arbitrarily many messages sent from one node to another. Hence, a network partition means that messages from the nodes in one component to another are dropped.

the lookup returns an identifier $i \in B$. We refer to the resulting configuration after all operations O_A and O_B as D. We now construct an execution starting in C, in which no partition takes place, where all the operations O_A take place first, and thereafter all the operations O_B take place. The asynchrony in the network permits delaying all messages between the two components long after the operations O_A and O_B are finished, making it appear as if there is a network partition. This execution is indistinguishable from the one in which the network partitioned. Hence, the system will end up in configuration D. Configuration D gives inconsistent lookups, as lookups for x initiated by a node in A will result in a different answer than lookups for x initiated by a node in B. More precisely, $lookup_D(x,i) \neq lookup_D(x,j)$ for $i \in A$ and $j \in B$. Since there only exists one network component, this contradicts the existence of a system which gives the assumed guarantees. □

Note that the impossibility result shows that lookup consistency is not possible in an asynchronous network which partitions. But perhaps lookup consistency is possible in an asynchronous network with failures, but without partitions. We do not know the answer to this. But the following observation makes us pessimistic.

We use failure detectors to detect and recover from failures. Nevertheless, any algorithm which attempts to detect failures in an asynchronous network risks inaccurately suspecting the failure of a correct, albeit slow, node [26]. The reason for this is that if this was not the case, the failure detector could be used to solve the *consensus problem* in an asynchronous network with failures, which is known to be impossible to solve [46]. Hence, our system may very well behave as follows. Assume a correct ring consisting of the nodes 0, 1, 2, 3, 4, and 5. At some point, node 2 inaccurately suspects that its predecessor 1 has failed, and node 4 inaccurately suspects that its successor 5 has failed. Similarly, node 1 inaccurately suspects its successor 2 has failed, and node 5 inaccurately suspects its predecessor 4 has failed. The system has partitioned into two parts, one containing $\{0,1,5\}$, and one containing $\{2,3,4\}$. Hence, our system will end up in the counter example used by the proof of the impossibility result. Note, that the mimicking of a network partition when using inaccurate failure detectors can occur in other topologies than the ring topology. Assume that such mimicking can be long lasting, and assume that availability requires a response before the ostensible partition recovers. Then a direct consequence of the theorem is that it is impossi-

ble to build a system which always guarantees lookup consistency and availability with inaccurate failure detectors.

As consequence of this, our goal will be to provide *eventual* lookup consistency in the presence of failures. Thus, we cannot guarantee lookup consistency when failures are detected, but as the network eventually becomes quiescent, we provide lookup consistency.

3.5.1 Periodic Stabilization and Successor-lists

In this subsection we show how the atomic ring maintenance for joins and leaves is modified to handle failures. This work relies on much work previously done by the authors of Chord. For a thorough reference, please refer to the Chord technical report [135].

The goal of periodic stabilization is to ensure that the pointers always eventually form a correct ring. However, the algorithms we have described make use of locking to guarantee lookup consistency. The atomic ring maintenance algorithms can therefore block if a node fails. Hence, we propose small modifications to the algorithms. Our goal will be to ensure that every lock in the system is eventually released. Periodic stabilization will take care of the rest, by ensuring that a correct ring is eventually formed. Hence, the system will eventually form a correct ring and all locks will eventually be released.

Next, we shortly describe the Chord protocols for periodic stabilization and the maintenance of successor-lists, and thereafter show our modifications.

Periodic stabilization, as we described in Section 2.3.4, has two purposes: incorporate new nodes into the ring and remove failed nodes from the ring. It, however, does that by relying on successor-lists, as we described in Section 2.3.4. But the successor-lists themselves may be incorrect due to joins and leaves, making the actions of periodic stabilization erroneous. For example, if a node p detects that its successor has crashed, it replaces it with the first alive entry q in its successor-list. Since the successor-list might be out of date, some node other than q might be the true successor of q. Hence, stabilization is done periodically to ensure that the ring is *eventually* correct.

The periodic stabilization protocol achieves its goals by striving to ensure that $p.succ.pred = p$ for any node p. This is done by two mechanisms: the *FixSucc* mechanism and the *FixPred* mechanism.

Informally, the *FixSucc* mechanism periodically moves the successor pointer of a node to the closest alive node in clockwise direction. This is partly achieved by the conditional of the STABILIZE procedure in Algorithm 1, which updates the *succ* pointer at p if it finds that the successor's *pred* pointer points to a closer node than p's current *succ* pointer.

Informally, the *FixPred* mechanism periodically moves the predecessor pointer of a node to the closest alive node in anti-clockwise direction. This is partly achieved in the conditional of the NOTIFY procedure in Algorithm 1, which updates the *pred* pointer at q if it finds that a node whose *succ* pointer is pointing at q is closer than q's current *pred* pointer.

Algorithm 1 does not suffice to achieve a correct ring in presence of leaves and failures, because the listed algorithms only ensure that a node points to the closest node, not to the closest *alive* node as required. For example, assume a system with correct pointers where node 10's successor is node 20, whose successor is node 30. If node 20 leaves the system or fails, the STABILIZE procedure at node 10 will fail to contact its successor to change its *succ* pointer to 30. This is therefore remedied as follows. If a node detects that its successor is no longer present, it replaces it with the first alive entry, f, in its successor-list. Even if f is not the correct successor of that node, the *FixSucc* mechanism will update *succ* such that it eventually points to the closest successor.

The above amendment will not ensure that the *pred* pointer always points to the closest alive predecessor. For example, assume a system with correct pointers where node 10's successor is node 20, whose successor is node 30. Assume node 20 leaves the system or fails, and the *FixSucc* mechanism correctly updates 10's *succ* pointer to 30. Next time node 10 invokes the NOTIFY procedure at node 30, the conditional will fail and node 30's *pred* pointer will continue to point at node 20. This is remedied by setting the *pred* pointer to `nil` if it is detected that the predecessor is no longer present. The conditional in the NOTIFY procedure is changed such that the *pred* pointer is always updated if it has value `nil`.

The successor-list at each node is maintained periodically as well. Every node periodically makes sure that its successor-list gets updated by copying its successor's successor-list, prepending the successor in the beginning of the successor-list, and truncating the list to a fixed size.

The above described *FixSucc* and *FixPred* mechanisms, as well as the maintenance of the successor-lists, are listed by Algorithm 11.

In the Chord technical report [135], it is shown that periodic stabi-

Algorithm 11 Periodic stabilization with failures

```
 1: procedure n.CheckPredecessor(p)          ▷ Locally called periodically
 2:     if IsAlive(pred) = false then
 3:         pred := nil
 4:     end if
 5: end procedure

 6: procedure n.Stabilize()                  ▷ Locally called periodically
 7:     try
 8:         p := succ.GetPredecessor()
 9:         if p ≠nil and p ∈ (n, succ] then
10:             succ := p
11:         end if
12:         slist := succ.GetSuccList()
13:         succlist := succ + slist                ▷ Prepend succ to slist
14:         succlist := trunc(succlist, k)   ▷ Right-truncate to fixed size k
15:         succ.Notify(n)
16:     end try catch(RemoteException)
17:         succ := getFirstAliveNode(succlist)     ▷ Get closest alive node
18:     end catch
19: end procedure

20: procedure n.GetPredecessor()
21:     return pred
22: end procedure

23: procedure n.GetSuccList()
24:     return succlist
25: end procedure

26: procedure n.Notify(p)
27:     if pred = nil or p ∈ (pred, n] then
28:         pred := p
29:     end if
30: end procedure
```

lization ensures that any interleaved sequence of joins and leaves will eventually result in a ring where $p.succ.pred = p$. For self-sufficiency, we include some of those theorems.

Theorem 3.5.2 (from [135]). *If any sequence of join operations is executed interleaved with stabilizations, then at some time after the last join the succ pointers will form a cycle on all the nodes in the network.*

The above theorem can be extended to *pred* pointers as well.

Corollary 3.5.3. *If any sequence of join operations is executed interleaved with stabilizations, then at some time after the last join the pred pointers will form a cycle on all the nodes in the network.*

Proof. *By Theorem 3.5.2 the succ pointers will form a cycle on all the nodes in the network. The* NOTIFY *procedure just maintains the invariant that if a node p correctly points at its successor q, then q's pred pointer will point back at p. Hence, the pred pointers will also form a cycle on all nodes in the network.* □

The size of the successor-list is usually set to be $\log_2(n)$, where n is the number of nodes in the system. Since, n is not globally known, it is either estimated or sometimes set to be the maximum number of nodes that could exist at any given time ($n = 2^{32}$ for every IP address). The reason for this is that it is proven that even if nodes would fail with probability 0.5, every node would still have some alive node in its successor-list. This result is proven, to varying degree of rigor, elsewhere [135, 72]. Hence, with an adequate size of successor-lists, the system remains connected in the presence of failures.

Theorem 3.5.4 (from [135]). *If we use a successor-list of length $r = O(\log N)$ in a network where every successor-list is correct, and then every node fails with probability $1/2$, then with high probability a lookup returns the closest living successor to the query key.*

We note that it is theoretically possible to construct a *loopy ring*, where $u.succ.pred = u$ for every node u, but where there exists a node v with an identifier between u and $u.succ$ (see Chapter 5). Periodic stabilization cannot rectify such a ring. But since its not known how such a loopy ring can occur, we ignore it in the rest of this chapter.

3.5.2 Modified Periodic Stabilization

Previous section showed that the periodic stabilization algorithm, with the *FixSucc* and *FixPred* mechanisms, handles both joins and failures. But the atomic ring maintenance already takes care of joins and leaves. Therefore, a viable question is whether a simpler algorithm than periodic stabilization, which only deals with failures, can be used in conjunction with atomic ring maintenance. Nonetheless, any algorithm which attempts to detect failures in an asynchronous network risks inaccurately suspecting the failure of a correct, albeit slow, node. Hence, in addition to atomic ring maintenance, the system needs to detect and recover from failures, as well as incorporate nodes which have been inaccurately classified as failed. Thus, we will use both the *FixSucc* and *FixPred* mechanisms of periodic stabilization.

The atomic ring maintenance algorithms will block if a node fails before the algorithm has terminated. The reason for this is that locks acquired by failed nodes will never be released. We propose a simple solution, which ensures that all locks eventually get released. Our first assumption is that periodic stabilization is run whenever a node's lock is free. Similarly, a precondition for the n.NOTIFY procedure is that node n's lock is free, otherwise it will not modify its *pred* pointer.

Before we describe how to deal with failures, we describe the philosophy behind it. Rather than checking whether a predecessor or a successor has failed, we use timers which when expired lead to the locks being released. In other words, locks are only leased for a certain amount of time. The reason why we use leased locks is that it guarantees that the locks are eventually released. There are several pitfalls in relying on detecting the failure of a successor or predecessor, rather than using timeouts as we propose. One reason is that a predecessor or successor might be alive, even though it never sends the final message that releases the lock. The reason for this could be a bug in the program. Moreover, it is not difficult for an adversary to make a client which acquires a lock, which it never releases.

Since we are using timeouts, it could always be that a timeout is premature, which results in several different join and leave operations getting intertwined. For example, some node might preemptively release a lock it is hosting because of a timeout. Thereafter, its lock might be acquired by some other node. By that time, the node which in the first case acquired

the lock might send, unaware of the preemptive release, some message according to the algorithm, which affects the latter operation. Therefore every node should always have as a precondition that the received message is in accordance with its lock. For example, a NEWSUCCACK message should always be ignored if the lock is free.

Furthermore, each joining and leaving node always attaches a random number to their leave or join operation. We refer to this as the *operation number*. This number is piggy-backed in all messages that have to do with the join or leave operation. Whenever the lock hosted by a node is acquired, the hosting node stores the operation number in a *opnum* variable. Whenever a node receives a message while its lock is not free, it ensures that *opnum* is equal to the operation number in the message, otherwise the message is ignored.

The join algorithm is modified, such that the successor of a joining node also piggy-backs its successor-list with the JOINPOINT message, such that the joining node can initiate its own successor-list.

Our goal is to ensure that a node whose lock is acquired, ensures that its lock is eventually released. This is achieved by every node i starting a timer as soon as the lock it is hosting, L_i, is acquired. The timer is turned off as soon as L_i becomes free. If the timer expires, the node simply changes the state of its lock to free, and sets *JoinForward* and *LeaveForward* to `false`. If a joining node's timer expires and $succ = \texttt{nil}$, then it restarts the join procedure until it gets its successor pointer. If a leaving node's timer expires, it simply leaves the system unnoticed.

We believe that the above algorithm will ensure eventual lookup consistency, which we motivate informally in the following. If no timeouts occur, the system will be the one described without periodic stabilization, and hence will provide lookup consistency. Hence, we turn to the case were timeouts occur. Because of timeouts, every lock is eventually released and the *JoinForward* and *LeaveForward* variables are set to `false`. This has two consequences. First, the node will start periodic stabilization. Second, it will ignore any remnant messages from any interrupted join or leave operation. If a timeout occurs, it either occurs at the successor of a joining or leaving node.

If a timeout occurs at the successor of a joining or leaving node, it will set its lock to free, making it start periodic stabilization. If the predecessor has indeed failed, periodic stabilization will recover from the crash failure, and the relevant locks will eventually be released, in which case

we are back to a correct system state, with guarantees lookup consistency. If the timeout is premature, and the predecessor is a leaving node, it will eventually timeout and leave unnoticed, which makes this case identical to the one where the predecessor indeed has failed. If the timeout is premature, and the predecessor is a joining node, periodic stabilization will eventually correct the joining node's *succ* pointer, provided that the joining node has a successor-list, which we assume it has acquired at the same time as it initially acquired its successor's address. Thereafter, the *FixPred* and *FixSucc* mechanisms will incorporate the new node into the ring.

If a timeout occurs at a joining node there are two cases, depending on if $succ = \texttt{nil}$. If the joining node has not set its *succ* pointer, which is required for periodic stabilization, it will restart the join and eventually get a correct successor. If $succ \neq \texttt{nil}$, all locks will eventually be released and periodic stabilization will incorporate the new node into the ring, since it has a *succ* pointer and a successor-list.

If a timeout occurs at a leaving node, it will leave, making it effectively a failure. Eventually all locks will be released, and periodic stabilization will rectify all pointers pointing at the absent node.

3.6 Related Work

Li, Misra, and Plaxton [89, 88, 87] independently discovered a similar approach as us. The advantage of their work is that they use assertional reasoning to prove the safety of their algorithms, and hence have proofs that are easier to verify. Consequently, their focus has mostly been on the theoretical aspects of this problem. They assume a fault-free environment, and do not use their algorithms to provide lookup consistency. Furthermore, they cannot guarantee liveness, as their algorithm is not starvation-free.

In the position paper by Lynch, Malkhi, and Ratajczak [95], it was proposed for the first time to provide atomic access to data in a DHT. They provide an algorithm in the appendix of the paper for achieving this, but give no proof of its correctness. In the end of their paper they indicate that work is in progress toward providing a full algorithm, which can also deal with failures. One of the co-authors, however, has informed us that they have not continued this work. Our work can be seen as a

continuation of theirs. Moreover, as Li *et al.* point out, Lynch *et al.*'s algorithm does not work for both joins and leaves, and a message may be sent to a process that has already left the network [89].

The problem of concurrently updating linked lists and other data structures has been studied in the context of *lock-free* algorithms for *shared-memory multiprocessors* [139, 63]. In this context a data structure resides in the shared memory of a computer, but the individual processors strive to correctly update the data structure concurrently without using locks, while guaranteeing that some processor always makes progress in updating the structure. The context is, however, different, which has led us to believe that these results are not directly applicable to our problems. First, failures in such contexts imply that individual processors have failed, while the memory storing the data structure is intact. This is not the case in distributed systems, where the data structure is distributed over many nodes, each holding part of the data structure in their local memory. Furthermore, the mentioned research provides lock-free implementations of singly-linked lists, while our data structure is a doubly-linked list. We believe that this subtle difference significantly complicates the problem.

The dining philosophers' problem has been widely studied as we previously mentioned. A widely adopted solution to the problem is to use randomization as suggested by Lehmann and Rabin [82]. They propose that each philosopher randomly choose whether to first pick right or left fork. This solution can, however, lead to a deadlock when the system size is small, which is the case at some point for every DHT. For example, if there are two nodes in the system and both pick left fork first, there will be a deadlock.

4 Routing and Maintenance

In this chapter we show how the basic ring structure, presented in the previous chapter, can be augmented with extra pointers to make routing more efficient. We provide different lookup strategies and give algorithms that work in concert with atomic ring maintenance. Finally, we provide algorithms that ensure that routing failures never occur unless nodes crash.

The ring structure has poor performance in terms of worst case message complexity and time complexity. The worst case time complexity and message complexity are n for the ring structure, because in the worst case all of the ring needs to be traversed, or if the search can go in both clockwise and anti-clockwise direction, half of the nodes in the ring need to be traversed. Our extension will make the worst case time and message complexity $\log_k(n)$, where k is a configurable constant, and n is the number of nodes in the system. This will in turn require that nodes carry additional routing tables of size $(k-1)\log_k(n)$. From now on we will refer to k as the *base of the system*.

4.1 Additional Pointers as in Chord

We now describe a simple extension to the ring, which will give us time and message complexity $\log_2(n)$ for n nodes. This extension is taken directly from the Chord system [136].

Each node maintains a *pred* pointer, a *succ* pointer, and a successor-list. In fact, the *succ* pointer of node p is pointing to the first node met going on the ring in clockwise direction starting at $p \oplus 1$. Hence, the *succ* pointer of p is pointing to the successor of the identifier $p \oplus 1$. A

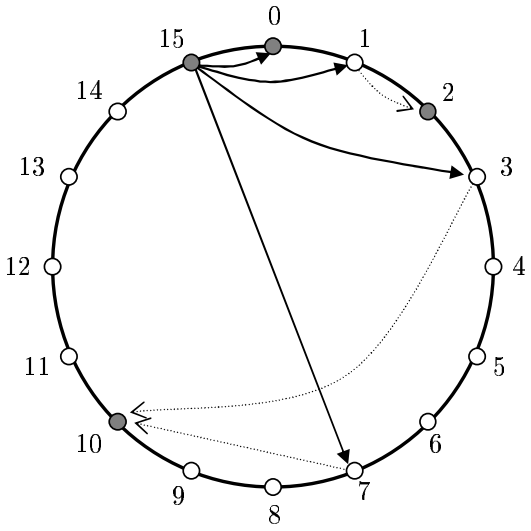

Figure 4.1: Simple extension of the ring with $\log_2(n)$ extra pointers. The filled circles indicate a node. The figure shows node 15's additional pointers.

simple extension is to let node p also point to the successor of $p \oplus 2, p \oplus 2^2, \cdots, p \oplus 2^{L-1}$, where $L = \log_2(N)$, where N is the size of the identifier space.

Figure 4.1 shows a system with an identifier space $\{0, 1, \cdots, 2^4 - 1\}$ ($L = 4$) and nodes $0, 2, 10, 15$. The figure shows node 15's additional pointers. Node 15 points to the successors of the identifiers $15 \oplus 2^0 = 0$, $15 \oplus 2^1 = 1$, $15 \oplus 2^2 = 3$, and $15 \oplus 2^3 = 7$. Note that several pointers might have the same successor, e.g. node 10 is the successor of both identifier 3 and 7 in Figure 4.1.

A node therefore has a routing table of size $\log_2(N)$, where N is the size of the identifier space. However, since nodes are spread uniformly across the ring, it can be shown that only $\log_2(n)$ entries are unique, where n is the number of nodes in the system. The number of unique pointers is significant, as it denotes the number of routing neighbors that need topology maintenance (discussed in Section 4.5).

4.2 Lookup Strategies

Lookups on the ring can now make use of more pointers. Before describing the exact lookup algorithm, we describe three lookup strategies that are applicable to every DHT:

- *Recursive lookup*
- *Iterative lookup*
- *Transitive lookup*

The first two lookup strategies are most common and can be traced back to DNS [107] [34, pg 5] [121, pg 3]. We define what we mean by each, and discuss their advantages and disadvantages.

We start by generalizing our description of a lookup, such that we can give algorithms for each lookup strategy and for different DHT operations such as put and get. An *initiating node*[1] starts a lookup to a particular *destination identifier* and some *operation*. The lookup algorithm will then route to the node responsible for the destination identifier, whereafter the responsible node performs the operation and returns the result of the operation back to the initiating node.

One particularly useful operation is to let the responsible node return its own contact information. In that case, the lookup simply returns the responsible node for a given destination identifier. The initiator can then implement basic DHT operations such as get, put, and delete, in a two-step scheme. First, the initiator makes a lookup to find the node responsible for a particular key. Thereafter, the initiator directly communicates with the responsible node to implement the desired operation. This approach has, however, the disadvantage that between the two steps, dynamism can affect the operation. For example, the responsible for the key might change between the two steps, or the responsible node might leave after the first step, making it necessary to restart the lookup. Another approach is to integrate the desired operation with the lookup. For example, the lookup can be used to implement a DHT get operation, where the responsible node returns the values associated with a key.

Every lookup algorithm can be defined in terms of two main abstractions: TERMINATE(i) and NEXT_HOP(i). The former is a boolean function

[1] We sometimes refer to the initiating node as the *initiator*.

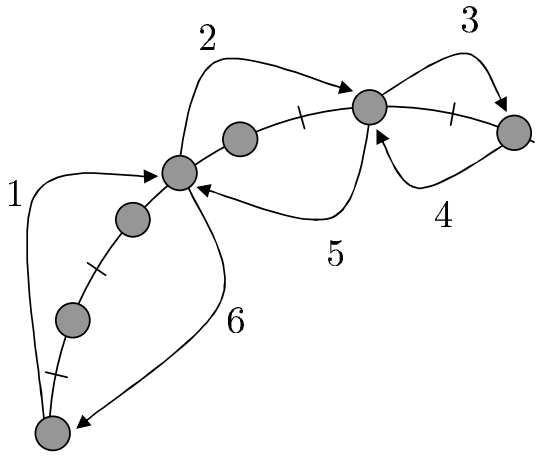

Figure 4.2: An illustration of recursive lookup. When a node receives a request, it either has the answer and returns it, or it asks its next hop for the answer and waits for a reply before responding to the requester.

that takes the destination identifier and returns true if the current node has the result of the lookup and wants to terminate the lookup. Otherwise the boolean function returns false. The NEXT_HOP(i) function takes the destination identifier and returns the next hop node in the routing process. Most importantly, if TERMINATE(i) is true, then NEXT_HOP(i) returns the address of the node responsible for i.

4.2.1 Recursive Lookup

When performing a recursive lookup, each node in the routing process recursively asks the next hop node for the node responsible for the destination identifier and returns whatever the next hop node returns. This process is described by Algorithm 12 and illustrated by Figure 4.2.

The obvious disadvantage of this approach is that every node in the path to the destination will be visited twice. Once as the query is being forwarded, and once when the result is being passed back. Hence, the probability of one of the nodes in the path leaving or failing increases, compared to iterative or transitive lookup.

If recursive lookup is combined with other operations, it can have performance drawbacks. For example, recursive lookup can be combined

with a DHT get operation, such that it returns the value associated with the identifier rather than returning the responsible node for the identifier. In this case, the value of the get operation has to travel through every node on the lookup path. In some applications, the values might be of substantial size and will considerably increase the overall latency and bandwidth consumption.

Algorithm 12 Recursive lookup algorithm

```
1: procedure n.LOOKUP(i, OP)
2:     if TERMINATE(i) then
3:         p :=NEXT_HOP(i)
4:         res := p.OP(i)                    ▷ OP could carry parameters
5:         return res
6:     else
7:         m :=NEXT_HOP(i)
8:         return m.LOOKUP(i, OP)
9:     end if
10: end procedure
```

There are, however, several advantages with recursive lookup compared to the other lookup strategies. The advantages have to do with the fact that nodes only communicate with the neighbors in their routing tables. Hence, nodes can use connection-oriented communication, such as TCP/IP, to maintain a connection with every routing neighbor. Hence, the lookup will be passed through connections which have been established in advance. This can reduce the latency of a lookup, as the cost of connection establishment is avoided. The cost of connection establishment includes detecting and rectifying the situation when a connection to another node cannot be established due to outdated references, firewalls, or NATs. Furthermore, sometimes a connection cannot be established to another node due to non-transitivity in the network, whereby a node p can establish a connection with q, and q can establish a connection with r, but node p cannot directly establish a connection with node r [48].[2]

In contrast to iterative lookup, the perhaps most important advantage of recursive lookup is that the system can employ proximity neighbor

[2]On the Internet, this phenomenon could be caused because one of the routers on the route between p and r is malfunctioning.

selection (see Chapter 1), where each node chooses to establish connections to nodes that it has low latency to, or it keeps only such nodes in its routing tables. Consequently, recursive lookup yields a low stretch value.

Reliable Recursive Lookup

It is more difficult to provide reliability for recursive lookups compared to iterative lookups. The difficulty lies in how to detect and recover from failures. A central question is whether every node on the lookup path should do failure detection or if that should only be done by the initiator.

In the former case, every node on the lookup path does failure detection on its next hop node. If it detects a failure, it removes that node from its routing table, and issues a new lookup, the result of which it returns to the caller. This requires that nodes remember pending lookups, such that they can reissue them.

It is important to not rely on timers which expire if the lookup response does not come back on time. The reason for this is that it is difficult to determine the right timeout value. Given a recursive lookup that goes through the nodes $x_1, \cdots, x_n$, the time it takes for x_i to receive a response is strictly higher than the time it takes for x_{i+1} to receive a response. Hence, each node on the lookup path needs to set a higher timeout value than its next hop node. Furthermore, a single node failure can cause a timeout on multiple nodes involved in the same lookup. These problems can be avoided by using failure detectors that use timers on heartbeat messages. Hence, no timing assumptions are made on the time it takes to receive a lookup response.

The inaccuracy of failure detectors can result in a node erroneously suspecting a failure and reissuing a lookup. It is therefore possible that multiple lookups are issued, leading to multiple responses. Hence, the initiator needs to filter redundant responses. The initiator does that by associating a unique identifier with every lookup request, and putting it in a pending set. The initiator removes the identifier of a lookup from its pending set whenever it receives a response for it. The initiator simply ignores any responses for identifiers that are not present in its pending set. Hence, redundant messages are filtered.

The other approach to reliable recursive lookups is to only let the initiator use a timer, which expires if too much time has passed without receiving a lookup response. If the timer expires, the initiator reissues

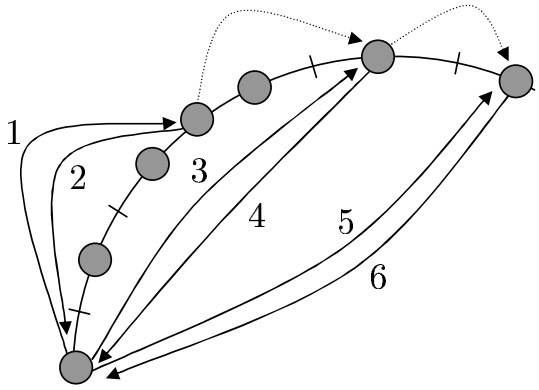

Figure 4.3: An illustration of iterative lookup. The initiator directly contacts every node on the path of the query until it receives the answer.

the lookup. The initiator might receive redundant lookup responses due to premature timeouts. Redundant lookup responses can be filtered using the same method as described above. One disadvantage of this approach is that it is difficult to estimate the expire time for the timer, as it depends on many variables, such as the system size. Nevertheless, this approach follows the end-to-end argument [125], which is how reliability is implemented on the Internet.

4.2.2 Iterative Lookup

With iterative lookup, the initiator contacts the first hop in the lookup path and receives back the address of the second hop node. Thereafter it contacts the second hop node and asks it for the third hop node, and so on, until it finds the node responsible for the destination identifier. This process is described by Algorithm 13 and illustrated by Figure 4.3.

The advantages and disadvantages of iterative routing are complementary to those of recursive routing. In contrast to recursive routing, nodes not only communicate with nodes in their routing table, but with many other nodes as well. There are several drawbacks to this, including problems related to establishing a connection or non-transitivity. Furthermore, proximity neighbor selection becomes pointless, because node p might not have a low latency to node r even though node p has low latency to q and q has low latency to r. It is, however, possible to

Algorithm 13 Iterative lookup algorithm

```
1: procedure n.LOOKUP(i, OP)
2:     m := n
3:     while not m.TERMINATE(i) do
4:         m := m.NEXT_HOP(i)
5:     end while
6:     p := m.NEXT_HOP(i)
7:     return p.OP(i)
8: end procedure
```

achieve some proximity awareness by using synthetic coordinates (see Section 1.2.2), which enables node p to approximate its latency to any node r.

One advantage of iterative routing is that the initiator can make parallel lookups, using multiple paths to the node responsible for the destination identifier. This is done in Kademlia [101] and EpiChord [83]. Hence, the initiator may be connected to several first hop nodes, and from them receive a list of candidate second hop nodes, from which it chooses a subset to communicate to, and so on. This way, the initiator can ensure that there is only a constant number of nodes involved in any parallel lookup. This approach has two advantages. First, only the nodes that first respond are chosen, which improves the latency. Second, it is resilient to individual node failures. Parallel lookups are generally not possible with the two other lookup strategies. We show, however, how it can be done in conjunction with replication (see Chapter 6).

Reliable Iterative Lookup

It is straightforward to implement reliable lookup with iterative routing. Since the initiator is involved in every step of the lookup, it can use a failure detector in every step of the algorithm. If a node fails, the initiator can reissue a lookup to another node. Note that the failure detector can use a timer on the expected lookup response. Unlike the failure detector used for recursive lookup, it is not necessary to use a heartbeat mechanism in the implementation of the failure detector. Redundant messages, which are generated due to the inaccuracy of failure detectors, can be avoided using the same technique as we described for implementing reliable re-

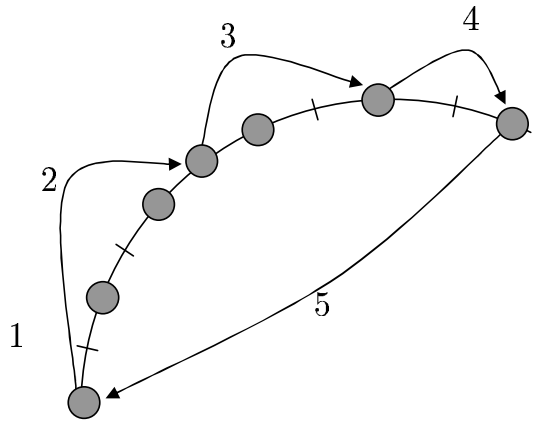

Figure 4.4: An illustration of transitive lookup. Every node delegates the responsibility of finding the responsible node to its next hop node. The node that knows the answer directly responds back to the initiator.

cursive lookup (see Section 4.2.1).

4.2.3 Transitive Lookup

Transitive lookup is similar to recursive lookup, but rather than passing back the result along the same path as the lookup, the result is directly sent back from the node terminating the lookup to the initiating node. This process is described by Algorithm 14, which partly contains event-based communication. Figure 4.4 illustrates a transitive lookup.

Transitive lookup is a hybrid of recursive and iterative lookup. It shares the advantage of recursive routing that nodes only communicate with nodes they are pointing to. An exception is the last step, in which the responsible node returns to the initiating node. This last step can suffer from all the problems we mentioned with iterative lookup. For example, NATs, firewalls, or non-transitivity in the network, can make communication with the initiating node impossible.

Aside from potential problems with the last routing step, transitive lookup benefits if proximity neighbor selection is used. Furthermore, transitive lookup avoids the latency and potential failures which recursive lookup suffers from when passing the result back along the lookup path. If transitive lookup is combined with a DHT get operation, it will

Algorithm 14 Transitive lookup algorithm

```
1:  procedure n.LOOKUP(i, OP)
2:      sendto n.LOOKUP_AUX(n, i, OP)
3:      receive LOOKUP_RES(r) from q
4:      return r
5:  end procedure

6:  event n.LOOKUP_AUX(q, i, OP) from m
7:      if TERMINATE(i) then
8:          p := NEXT_HOP(i)
9:          sendto p.LOOKUP_FIN(q, i, OP)
10:     else
11:         p :=NEXT_HOP(i)
12:         sendto p.LOOKUP_AUX(q, i, OP)
13:     end if
14: end event

15: event n.LOOKUP_FIN(q, i, OP) from m
16:     r := OP(i)
17:     sendto q.LOOKUP_RES(r)
18: end event
```

avoid the overhead of passing the return value through every node on the lookup path.

Reliable Transitive Lookup

Reliable transitive lookup can be implemented using the end-to-end approach described for reliable recursive lookup (see Section 4.2.1). It is much more complicated to let every node in the lookup path use failure detectors and reissue lookups. The difficulty is that a node does not know when to stop reissuing lookups. In reliable recursive lookup, a node only reissues a lookup if it has a pending request for which it has not yet received a response. In the transitive lookup, only the initiator receives a response. Hence, the other nodes in the lookup path do not know if they should reissue a lookup after they detect a failure, or if the lookup has terminated correctly.

4.3 Greedy Lookup Algorithm

We now describe how *greedy routing* is done to find the successor of an identifier, and hence the responsible node. Whenever a node p receives a lookup for *destination identifier* i, it checks whether its successor is responsible for that identifier, in which case it terminates the lookup. Otherwise, it tries to forward the request to the pointer in the range (p, i), which is closest in clockwise direction to i. Put differently, it tries to forward the request to the closest possible node without *overshooting*[3] the destination identifier. If there is no such closest node, that means that the successor of the current node will be the successor of the destination identifier i. Hence, the last step in the lookup path uses the successor pointer of a node. Algorithm 15 shows the corresponding implementation of TERMINATE(i) and NEXT_HOP(i).

The routing table of a node p, together with its *succ* and *pred* pointers, are represented by a monotonic function rt, which maps integers to node identifiers. Therefore, $rt(1)$ points to the successor of p, and $rt(2)$ points to the second closest node, in clockwise direction, in p's routing table, etcetera. Hence, if p has K pointers, $rt(K)$ points to p's predecessor, which is the node farthest away from p in clockwise direction.

[3]We say that a node p overshoots an identifier i if p routes to j when $d(p, i) \leq d(p, j)$.

Algorithm 15 Greedy lookup

```
1: procedure n.TERMINATE(i)
2:     return i ∈ (n, succ]
3: end procedure

1: procedure n.NEXT_HOP(i)
2:     if TERMINATE(i) then
3:         return succ
4:     else
5:         r := succ
6:         for j := 1 to K do                ▷ Node has K pointers
7:             if rt(j) ∈ (n, i) then
8:                 r := rt(j)
9:             end if
10:        end for
11:        return r
12:    end if
13: end procedure
```

A few things can be noted about the above algorithm. An invariant of this algorithm is that the lookup request will always reach the predecessor of the destination identifier and then be sent to the successor of the destination identifier. Consequently, if a lookup already starts at the successor of the destination identifier, it will be routed back through the predecessor of the initiator before terminating.

We shortly summarize the following work previously done on Chord [135]. It has been proven that at each step in the routing process, the distance, in the identifier space, to the destination identifier will be halved. Hence, the successor of an identifier will be found in maximum $\log_2(N)$ hops, where N is the size of the identifier space,. This, however, can be a quite large number as the number of nodes, n, is often much smaller than the size of the identifier space. By assuming that nodes are distributed uniformly on the ring, it has been proven that, with high probability, the worst case number of hops to reach the destination is $2\log_2(n)$ hops, where n is the number of nodes. In summary, lookups can be performed in $O(\log n)$ time, where n is the number of nodes.

4.3.1 Routing with Atomic Ring Maintenance

In the previous chapter we described how atomic ring maintenance could be used to ensure lookup consistency on the ring. In this chapter we have provided a different routing algorithm which not only routes on the ring, but also uses the additional pointers in the system. This routing algorithm can be integrated with the atomic ring maintenance algorithms to ensure lookup consistency.

The key to providing lookup consistency is in the invariant that lookups always pass through the predecessor of the responsible node. Hence, the last hop of any lookup uses the *succ* pointer of the penultimate node. If atomic ring maintenance is implemented as described in Section 3.3, the last hop can simply use the *succ* pointer as normal. The final node should always ensure two things depending on its state. If its *JoinForward* flag is enabled, it should forward the request to its predecessor. Otherwise, if its *LeaveForward* flag is enabled, it should send the lookup to its successor. This way, lookup consistency will be guaranteed as proved in Section 3.3.

4.4 Improved Lookups with the k-ary Principle

We next show how the pointers can be placed to achieve a time complexity of $\log_k(n)$, where n is the number of nodes and the base k is some predefined constant. We refer to this as doing k-ary lookup or placing pointers according to the k-ary principle. As we mentioned in Chapter 1, this can be practical, as setting $k = N^{\frac{1}{r}}$ guarantees a worst case lookup of r hops, where r can be chosen to be any positive integer. This of course comes at the cost of increased routing tables, which in turn requires maintenance as nodes join and leave. In some applications, however, this compromise is feasible.

To achieve k-ary lookup, we assume that the size of the identifier space is a power of the desired base k, i.e. $N = k^L$ for some integer L. Each node, in addition to storing *succ* and *pred* pointers, maintains a routing table. The routing table consists of $L = \log_k(N)$ *levels*. At each level l ($1 \leq l \leq L$) a node p has a view of the identifier space defined as:

$$V_l = \left[p, p \oplus k^{L-l+1}\right)$$

This means that for level one, the view consists of the whole identifier space, because $V_1 = \left[p, p \oplus k^L\right)$. At any other level ($l > 1$), the view consists of one k:th of V_{l-1} space. Put differently, the first level view of node p consists of all identifiers. Level two's view consists of a subset of level one's identifiers, specifically the one k:th of the identifiers closest to node p. Level three consists of a subset of level two's identifiers, in particular the one k:th of the identifiers closest to node p.

At any level l ($1 \leq l \leq L$) the view is partitioned into k equally-sized intervals denoted I_i^l for $0 \leq i \leq k-1$. At a node p, I_i^l is defined as:

$$I_i^l = \left[p \oplus ik^{L-l}, p \oplus (i+1)k^{L-l}\right), \;\; 0 \leq i < k, \; 1 \leq l \leq L$$

Each node p maintains a *contact node* for each interval in its routing table. For simplicity, we will take the contact to be the successor of the beginning of the interval. But more flexible choices are also valid, such as any node in the interval as we describe in Section 4.5. Thus, for all intervals $j \in \{1, 2, .., k-1\}$, the successor for interval I_j^l is chosen to be the first node encountered, moving in clockwise direction, starting at the beginning of the interval. This implies that for any level l ($1 \leq l \leq L$) the

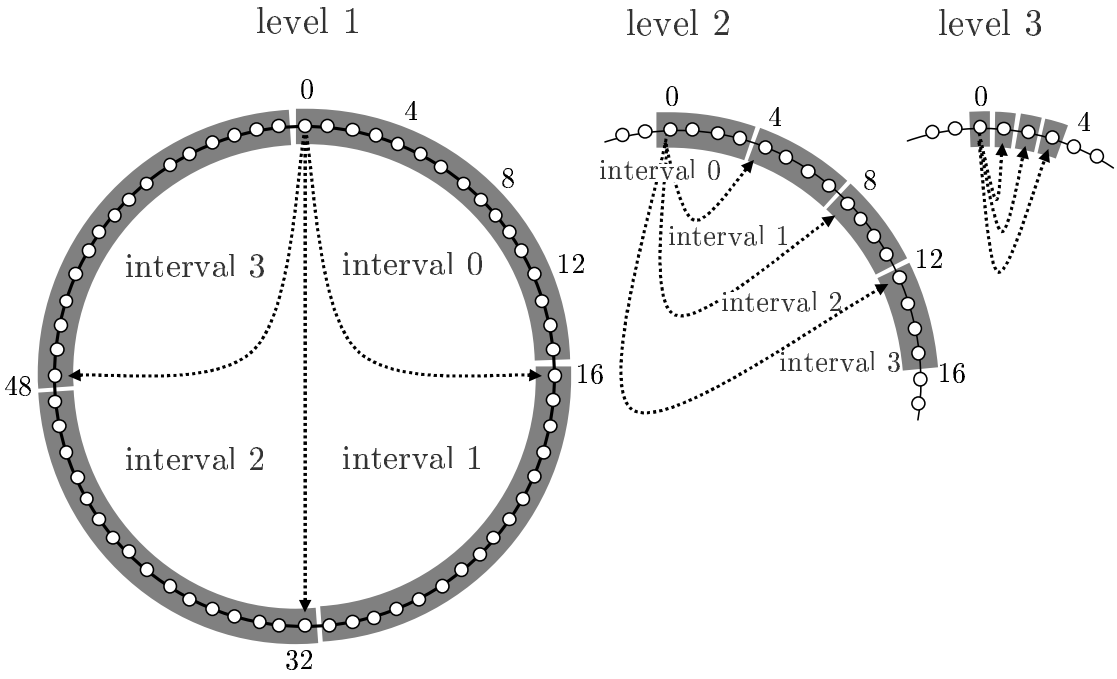

Figure 4.5: Figure of the routing table of node 0, for $N = 64$ and $k = 4$. The dotted arrows are the start of the intervals. The dark regions represent the respective intervals. The left most figure shows the intervals on level one. The center figure shows the intervals on level two. The right-most figure shows the intervals on level three.

successor for interval I_0^l is always p itself. We will use $S(I)$ to denote the identifier of the successor node for interval I.

Figure 4.5 shows how an identifier space of size $4^3 = 64$ is divided when the base $k = 4$. Hence, the space consists of 3 levels ($\log_4(64) = 3$) and each level is divided into 4 intervals ($k = 4$).

Illustrating Routing by Trees The above routing table is sufficient to achieve $\log_k(n)$ lookup hop counts, where n is the number of nodes and k is the base of the system.

Another way to represent the routing tables at each node is by a k-ary tree. Figure 4.6 shows the k-ary tree for node 10 when $k = 3$ and the identifier space is $\{0, 1, \cdots, 26\}$. For simplicity we assume a fully populated system, i.e. where there is a node for every possible identifier.

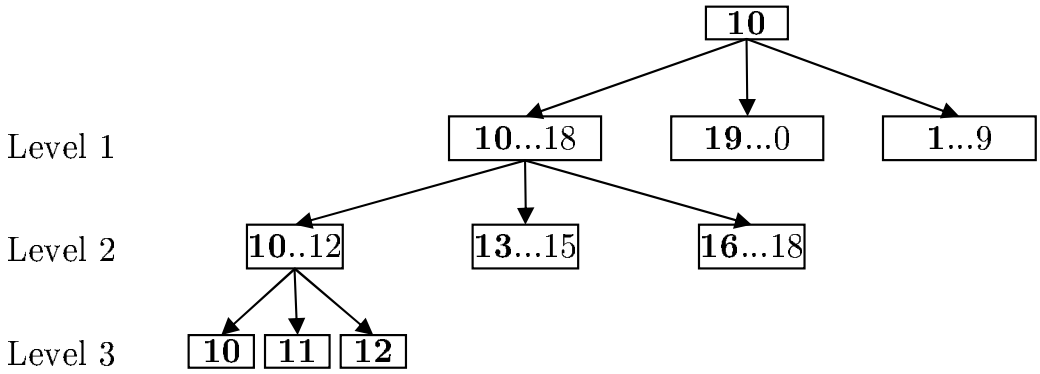

Figure 4.6: Node 10's k-ary tree when $k = 3$, and identifier space size is $3^3 = 27$. The system is fully populated. Vertices show an interval as well as the successor of the interval in bold.

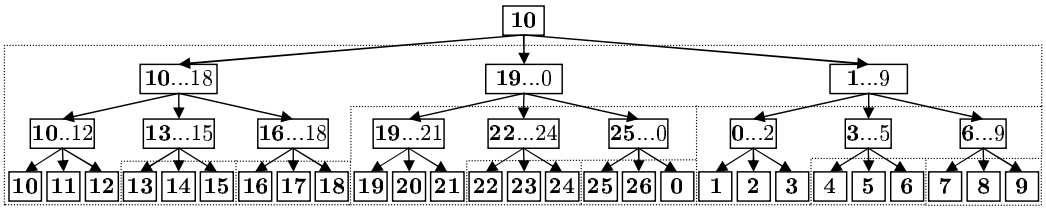

Figure 4.7: Virtual k-ary tree rooted at node 10, when $k = 3$, and identifier space size is $3^3 = 27$. The system is fully populated. The dotted rectangles indicate k-ary trees at different nodes. Vertices show an interval as well as the successor of the interval in bold.

Each vertex in the tree shows an interval as well as the successor of the interval in bold typeface.

It can be useful to extend the k-ary tree at a node into a *virtual k-ary tree* which shows how routing would proceed. Figure 4.7 shows the virtual k-ary tree for the same setting as in Figure 4.6.

Routing on the virtual k-ary tree The virtual k-ary tree shows the path of the lookup. Assume a lookup is initiated by node 10 for identifier 26 in the fully populated system depicted by the virtual k-ary tree in Figure 4.7. Node 10 uses its k-ary routing table and finds that node 19, which is the successor of interval [19...0], is its closest neighbor preceding 26. Hence, the request is routed to node 19. Node 19 would use its k-ary routing

table, and find that its closest neighbor preceding 26 is 25, which is the successor of interval [25..0]. Node 25 would finally forward the lookup to node 26, which is the successor of interval [26].

A virtual k-ary tree similar to the one in Figure 4.7 can be made for every system setting, including non-fully populated systems. Such a virtual k-ary tree is constructed from the actual k-ary routing tables of each node. Hence, if some node 10 has node 23 as successor for its interval $[22, 25)$, the sub-tree of vertex 23 would be node 23's routing pointers with 23's view of the intervals. The virtual k-ary tree is merely a logical construction to help understanding how routing works, not a structure which is represented and used for routing. For more details on this, please refer to our previous work on the topic [7].

Using the virtual k-ary tree we can now prove that the worst case lookup length is $2\log_k(n)$ with high probability, where n is the number of nodes and k the base of the system.

Theorem 4.4.1. *Lookup takes at most $2\log_k(n)$ hops with high probability where n is the number of nodes and k is the base of the system.*

Proof. *Routing proceeds in the k-ary tree, moving down one level in each hop. The k-ary tree consists of $\log_k(N)$ levels where N is the size of the identifier space.*

After t hops, where $t = 2\log_k(n)$, the size of the current interval I_j^t, for some j, will be

$$k^{L-t} = \frac{k^L}{k^t} = \frac{N}{n^2}$$

Assuming uniform distribution of nodes on the ring, the expected number of nodes in an interval of size $\frac{N}{n}$ is 1, hence the interval $\frac{N}{n^2}$ contains one node with probability $\frac{1}{n}$, which becomes negligible as n grows. Hence, with high probability the destination is reached within at most $2\log_k(n)$ hops.

□

Note that several of the routing hops can be local hops, as the successor of an interval I_0^l at a node p, for any l, is p itself.

4.4.1 Monotonically Increasing Pointers

It can sometimes be convenient to organize the pointers similarly to Chord. In other words, rather than having two dimensions, one for levels and

one for intervals, pointers are indexed sequentially such that at any given node, a pointer with a higher index always points farther away in the identifier space than a pointer with a lower index.

Instead of having pointers in levels and intervals, a node can keep $(k-1)\log_k(N)$ pointers, for some fixed base k where the size of the identifier space is $N = k^L$ for some positive integer L. Node p keeps a pointer to a contact node for the start of every interval $f(i)$, where $1 \leq i \leq (k-1)\log_k(N)$ where:

$$f(i) = p \oplus (1 + ((i-1) \mod (k-1)))\, k^{\lfloor \frac{i-1}{k-1} \rfloor}$$

The above two schemes are equivalent to each other, except that in the latter, intervals which would produce local hops have been removed.

Theorem 4.4.2. *The start of the interval I_i^l is equivalent to $f((L-l)(k-1)+i)$, for any level $1 \leq l \leq \log_k(N)$ and interval $1 \leq i < k$.*

Proof. *We abuse notation and let I_i^l denote the start of the interval it represents. We use the fact that adding any multiple of a number k does not affect the outcome when doing modulo k arithmetic.*

$$f((L-l)(k-1)+i) = p \oplus (1 + (((L-l)(k-1)+i-1) \mod (k-1)))\, k^{\lfloor \frac{(L-l)(k-1)+i-1}{k-1} \rfloor}$$

$$f((L-l)(k-1)+i) = p \oplus (1 + ((i-1) \mod (k-1)))\, k^{\lfloor \frac{(L-l)(k-1)+i-1}{k-1} \rfloor}$$

$$f((L-l)(k-1)+i) = p \oplus ik^{\lfloor \frac{(L-l)(k-1)+i-1}{k-1} \rfloor}$$

$$f((L-l)(k-1)+i) = p \oplus ik^{\lfloor L-l+\frac{i-1}{k-1} \rfloor}$$

$$f((L-l)(k-1)+i) = p \oplus ik^{L-l}$$

□

Chord's pointers are simply a special case of the way pointers are placed by the above scheme.

Corollary 4.4.3. *Chord's intervals are equivalent to the intervals $f(i)$ when $k = 2$.*

Proof. *We use the fact that if $k = 2$ then any integer modulo $k - 1$ is zero.*

$$f(i) = p \oplus (1 + ((i-1) \mod (2-1)))\, 2^{\lfloor \frac{i-1}{2-1} \rfloor}$$

$$f(i) = p \oplus (1) 2^{\lfloor \frac{i-1}{2-1} \rfloor}$$

$$f(i) = p \oplus 2^{i-1}$$

□

Just as $f(i)$ denotes the start of the interval, $rt(i)$ denotes the contact node for $f(i)$.

For more information on k-ary search in distributed hash tables, please refer to our previous work [7, 41, 5, 40].

4.5 Topology Maintenance

Up until now, we have not discussed the impact of dynamism on the system. As nodes join, leave, and fail, routing information becomes stale and needs to be updated. This section describes a method to efficiently maintain the routing information in the presence of dynamism. Chapter 3 already showed how to maintain the ring. Hence, the focus of this section is how to maintain the additional pointers described in this chapter. Topology maintenance concerns joins, leaves, and failures. Even though all three events are highly related, next section focuses on failures, while the subsequent section deals with joins and leaves.

4.5.1 Efficient Maintenance in the Presence of Failures

Additional routing pointers are discovered through lookups. Similarly, fault-tolerance of routing information is about detecting failed routing neighbors and replacing them with other nodes by making lookups. Another method of dealing with failures is through replication, but that is the topic of Chapter 6.

Initialization A joining node, which has been incorporated into the ring using atomic maintenance, still needs to populate the rest of its routing table according to the k-ary principle. The k-ary principle does not require that the successor of each interval is picked as a contact node, but rather any node in each interval can be kept as the contact node for that interval. Therefore, a joining node can initially populate its routing table by issuing lookups to the start of each interval for every additional routing entry, or it can use its successor's routing table to approximate its own routing Since the contact node does not need to be the successor of the start of the interval, the lookup can be used with an operation that returns the successor-list at the responsible node (see Algorithm 16). Thereafter, the joining node can pick any of those nodes as its contact node for that interval. In practice, it can probe a constant number of them and choose the one that it finds most suitable, in terms of some metric such as latency.

Algorithm 16 Routing table initialization

```
1: procedure n.InitRoutingTable()
2:     for i := 1 to (k − 1) log_k(N) do
3:         n.UpdateEntry(i)
4:     end for
5: end procedure

6: procedure n.UpdateEntry(i)
7:     S :=n.Lookup(f(i),GetSList())        ▷ f(i) as in Section 4.4.1
8:     rt(i) := s′                          ▷ s′ is the "best" node in S
9: end procedure

10: procedure n.GetSList()
11:     return {n} ∪ succlist               ▷ Return own id and successor list
12: end procedure
```

Fault-detection and Recovery A node will use an unreliable failure detector to detect if any of the additional routing pointers fail. This can be implemented by having each node periodically send a heartbeat message to each of its additional pointers $rt(i)$ and waiting to receive an acknowledgment. If the failure detector suspects that the node in routing entry

i has failed, it triggers the UpdateEntry event, shown in Algorithm 16, with parameter i.

The failure detector needs to be strongly complete, but we do not require it to be accurate (see Chapter 2). Since the failure detector has strong completeness, every failure will eventually be detected and replaced with another entry. However, the inaccuracy of the failure detector might trigger updates to entries which point to non-failed nodes. This does not affect the functionality of the system, but rather increases the amount of bandwidth used for topology maintenance. Increased accuracy lowers the excess bandwidth used for topology maintenance.

Other systems use periodic lookups to deal with failures. The reason why we suggest using failure detectors is to avoid the lookup cost, which often is $O(\log n)$. Hence, with our proposal, the cost of topology maintenance will be $O(1)$ per routing entry when there are no failures, rather than the typical $O(\log n)$, for an n node system.

A fundamental difference between our described topology maintenance mechanism and the ones used by other systems is that our does not always try to point to a contact node that is inside the interval for which it is a contact. This can be disadvantageous in certain scenarios. For example, assume the system consists of two nodes, one with identifier 0 and one with identifier 1, and the identifier space is $[0, 1023]$. Hence, all of 1's additional pointers point at 0, and vice versa. If another 1002 nodes join, our topology maintenance mechanism will not update any of node 0's or node 1's additional pointers. Note that this is only a problem if the contact node is outside the interval for which it is a contact. Therefore, we suggest using a hybrid approach, where failure detectors are used, which frequently send heartbeats, and less frequent periodic lookups are made for a pointer whenever the contact node for that interval has an identifier outside the interval.

4.5.2 Atomic Maintenance with Additional Pointers

We now describe how to integrate atomic ring maintenance with topology maintenance for joins and leaves. A subtlety with structured overlay networks is the potential of *routing failures* even in the absence of node failures. By a routing failure we mean sending to, or expecting to receive messages from, a neighbor that has left the system. We say a node q is a *neighbor* of a node p if q is in the routing table of p. The reason for this

is that nodes will continue to point to a node that left the system until their failure detectors discover that the node no longer exists. This process can take a substantial amount of time in an asynchronous network. Meanwhile, some lookups might attempt to use some of those dangling pointers for routing. Hence, even in the absence of node failures, routing can fail. This is true for most structured overlay systems, such as Chord [136], Pastry [123], and Bamboo [121].

Routing failures are defined in terms of neighbors. Some operations, such as lookup, send messages to other nodes than their neighbors. For example, the last message of a transitive lookup is sent from the responsible node to the initiator, even though the initiator might not be the responsible node's neighbor. The same applies to the RPC responses of recursive lookup. Nevertheless, it is possible to avoid transitive or recursive lookup failures in absence of node failures. This can be ensured by guaranteeing that a node does not leave the system until all blocking receive statements have terminated. Hence, the initiator of a transitive lookup does not leave the system until its blocking receive has terminated (Line 3 in Algorithm 14). Similarly, a node involved in a recursive lookup will not leave the system until its RPC call (Line 8 in Algorithm 12) has terminated. Note that RPC is implemented using blocking receive (see Chapter 2).

In this section we describe how to provide a system that does not exhibit any routing failures in the absence of node failures. Thus, we avoid the cost of fault-recovery when there are no failures. Achieving this is facilitated by atomic ring maintenance, as described in Chapter 3.

When a node joins the system, two things need to happen. First, the newly joined node needs to discover contact information for the nodes to which it wants to maintain additional routing pointers. Second, other nodes might need to modify their routing information, such that they point to the newly joined node. Regardless of how these two operations are done, we want nodes to know about the identity of other nodes pointing to them. Hence, if node p points to node q, node q should know that p is pointing to it. Every node therefore maintains a *backlist* containing a list of nodes pointing to it. The backlist enables a leaving node to notify other nodes to remove their pointers to it. We refer to the messages used to add and remove information from backlists and routing tables as *accounting messages*, and refer to all other messages, such as lookup messages, as *ordinary messages*.

The problem is seemingly simple. An algorithm, however, needs to account for all possible interleavings when two nodes that are either pointing to each other or are in each other's backlists, are leaving at the same time.

A question is whether the correctness property should be to guarantee no routing failures of ordinary messages in the absence of failures, or to guarantee no routing failures (of both ordinary and accounting messages) in the absence of failures. We present one solution for each of the correctness assumptions.

Simple Accounting Algorithm Assume that we want to guarantee no routing failures of ordinary messages, but allow routing failures of accounting messages when the system is free from node failures. Then the following *simple accounting algorithm* solves the problem. Our assumption of FIFO channels will be crucial for the correctness of the algorithm. Every routing table is represented by the set RT and each backlist by the set BL.

Whenever a node p is to add another node q to its routing table, the event AddRT(q) is triggered, which sends a message to q asking q to add p in its backlist, and node q responds with an acknowledgment. Only after receiving the acknowledgment, node p incorporates q into its routing table.

An algorithm similar to the one for adding nodes is used before leaving by triggering the event AccountLeave. If node p is leaving and q is in p's backlist, p sends a message to q asking it to remove p from its routing table. Node q then responds with an acknowledgment, whose receipt enables node q to leave. A counter c is used, which is initially set to zero, to keep track of the number of pending requests. After the last acknowledgment is received and, thus, $c = 0$, node p can leave the system.

Theorem 4.5.1. *The simple accounting algorithm (Algorithm 17) will ensure that there are no routing failures of ordinary messages in the absence of node failures.*

Proof. *The algorithm enforces the invariant that whenever a node q is in the routing table of p, node q will remain in the system. Node q will only appear in p's routing table after p gets the acknowledgment from q that q has put p*

Algorithm 17 Simple accounting algorithm

```
1: event n.AddRT(q) from app        ▷ Called when q is to be added to RT
2:     sendto q.AddBL()
3: end event

4: event n.AddBL() from m
5:     BL := BL ∪ {m}                                  ▷ BL is backlist set of n
6:     sendto m.AckAddBL()
7: end event

8: event n.AckAddBL() from m
9:     RT := RT ∪ {m}                                  ▷ RT is routing table set of n
10: end event

11: event n.AccountLeave() from app
12:     for p ∈ BL do
13:         sendto p.RemRTEntry()
14:         c := c + 1                                  ▷ c is initially 0
15:     end for
16: end event

17: event n.RemRTEntry() from m
18:     RT := RT − {m}                                  ▷ The entry can be replaced
19:     sendto m.AckRemRTEntry()
20: end event

21: event n.AckRemRTEntry() from m
22:     c := c − 1
23:     if c = 0 then
24:                                                     ▷ Leave the system
25:     end if
26: end event
```

in its backlist and, hence, that q is in the system. Similarly, if p has q in its routing table, q will only leave after p acknowledges that q is no longer in its routing table. The FIFO and reliability requirements enforce that every message sent from p will be received by q. In particular, the FIFO requirement ensures that the acknowledgment message for a leave from p to q "flushes" all outgoing ordinary messages from p to q. □

The algorithm is integrated with the atomic ring maintenance by performing the leave part of the accounting algorithm after the leave point is reached. The reason for this is that the atomic maintenance guarantees that no lookups will end up at the leaving node after the leave point has been reached. Thus, no new pointers will be created to the leaving node thereafter.

If node failures are introduced, the above algorithm will block. To deal with crash failures, we propose to use failure detectors when waiting for the acknowledgment messages, and proceed if the failure detector suspects that the sending node has failed. Then, the algorithm will always terminate. Inaccurate suspicions, however, can result in routing failures.

A drawback of the above algorithm is that the accounting messages are susceptible to routing failures even in the absence of node failures. For example, assume p has q in its routing table and that q, consequently, has p in its backlist. Moreover, assume q does not have p in its routing table. Then if p leaves the system, q will still have p in its backlist. If q later leaves, it will attempt to contact p, asking it to remove q from its routing table. Hence, this will result in a routing failure since node p is no longer in the system. Next, we strengthen the correctness assumption to avoid such situations.

Fault-free Accounting Algorithm We now present an algorithm to ensure no routing failures for ordinary messages, as well as accounting messages, in the absence of node failures. To achieve this, the algorithm increases the number of messages by a constant factor compared to the simple algorithm.

Algorithm 18 shows the fault-free accounting algorithm. Again we assume reliable communication and FIFO channels. The algorithm is an extension of the simple accounting algorithm. The algorithm can be augmented to handle node failures similarly to the simple accounting algorithm.

Joining is identical to the simple accounting algorithm. Whenever a node wishes to add a node to its routing table, it triggers the event AddRT with a parameter specifying the new node it wishes to add to its routing table. The event AddRT(q) at node p asks node q to add p to its backlist. After receiving the request, node q adds p to its backlist and responds with an acknowledgment. Only after receiving the acknowledgment, node p adds q to its routing table.

Leaving involves a few more operations than the simple accounting algorithm. Whenever a node p wishes to leave the system, it triggers the event AccountLeave, which iterates through every element in $RT \cup BL$, and sends the corresponding node q a RemEntry message. Moreover, if q is in p's RT, node p immediately removes it from there. The motivation behind this is that node p is leaving anyway, and will therefore not need to use that pointer. After q receives the request, it ensures that p does not appear in both its routing table and its backlist. Thereafter, it responds with an acknowledgment. A counter is used similarly as in the simple accounting algorithm.

We now prove the following safety property about the algorithm.

Theorem 4.5.2. *The fault-free accounting algorithm (Algorithm 18) is free from routing failures of ordinary and accounting messages assuming absence of node failures.*

Proof. *The fault-free accounting algorithm only extends the simple accounting algorithm, hence we know from Theorem 4.5.1 that the fault-free accounting algorithm is free from routing failures of ordinary messages. It remains to show that it is free from failures of accounting messages. Assume by contradiction that a routing failure occurs when p sends a message to q at time t. At time t, node p either had q in $BL \cup RT$ or it is responding back with an acknowledgment to q. We analyze each case separately.*

Case 1: p has q in $BL \cup RT$ at time t. Our assumption of reliable communication implies that q was no longer present at time t. Node q can only have left after it has received acknowledgments (AckRemEntry*) from all nodes that have q in their RT or BL. By the FIFO assumption, node p must have sent* AckRemEntry *to q before time t. Hence, p must have removed q from both its BL and RT before time t when the event* RemEntry *happened. This contradicts the occurrence of a routing failure since p cannot have pointed at q at time t.*

Case 2: p is responding with an AckRemEntry *to q. This case leads to a contradiction since p only sent* AckRemEntry *in response to a* RemEntry

implying that $c > 1$ *at* q*. Hence,* q *cannot leave before the message from* p *reaches* q*.* □

Algorithm 18 assumes uni-directional links, possibly with two uni-directional links in opposite directions between the same two nodes. If all links are bi-directional, the algorithm can be adapted by replacing the occurrence of BL with RT everywhere in the algorithm.

Algorithm 18 Fault-free accounting algorithm

```
1: event n.AddRT(q) from app      ▷ Called when q is to be added to RT
2:     sendto q.AddBL()
3: end event

4: event n.AddBL() from m
5:     BL := BL ∪ {m}                          ▷ BL is backlist set of n
6:     sendto m.AckAddBL()
7: end event

8: event n.AckAddBL() from m
9:     RT := RT ∪ {m}                          ▷ RT is routing table set of n
10: end event

11: event n.AccountLeave() from app
12:     for p ∈ RT ∪ BL do
13:         sendto p.RemEntry()
14:         c := c + 1                                  ▷ c is initially 0
15:         RT := RT − {p}                      ▷ No more messages to q
16:     end for
17: end event

18: event n.RemEntry() from m
19:     RT := RT − {m}
20:     BL := BL − {m}
21:     sendto m.AckRemEntry()
22: end event

23: event n.AckRemEntry() from m
24:     c := c − 1
25:     if c = 0 then
26:                                             ▷ Leave the system
27:     end if
28: end event
```

5 Group Communication

In this chapter, we show how the interconnectivity of the nodes in a DHT can be used for group communication. In other words, we provide algorithms which enable any node to send a message to all nodes on the ring. We also provide algorithms that enable any node to efficiently send a message to all nodes in a specified set of identifiers, e.g. broadcast a message to all nodes with identifiers in the set $\{3, 4, 21, 22\}$. Similarly, we provide algorithms that enable any node to efficiently send a message to the nodes responsible for any of the identifiers in a specified set of identifiers.

Motivation The lookup operation provided by DHTs is sometimes insufficient for some applications, since it is limited to finding keys which *exactly match* the provided query. The lookup operation does not provide for complex queries containing wildcard expressions, such as "find all items which have a key containing the keyword `music`". Nor does the lookup operation provide for queries containing regular expressions.

One solution to the above problems is to broadcast a query to all nodes, or some of the nodes, which are part of the DHT. In fact, many popular peer-to-peer applications — such as Skype, Kazaa, and Gnutella — build a network where the nodes are interconnected randomly. Queries are recursively flooded or broadcast to all neighbors. The lack of structure in the random networks, however, leads to disadvantages. For example, some nodes might receive the same message redundantly from different neighbors. Furthermore, the freedom of letting nodes randomly interconnect might lead to the network partitioning into several components which have no interconnection, even though all nodes are connected in the underlay network. Group communication on top of DHTs does not

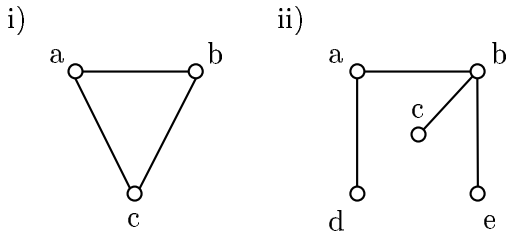

Figure 5.1: i) shows a graph with a cycle ii) shows a graph without any cycles.

suffer from these disadvantages [22, 42, 23]. On the contrary, when using DHTs, it is possible to guarantee that the time complexity is logarithmic to the number of nodes in the DHT, and that the message complexity is equal to the number of nodes in the DHT.

Group communication can be used as a basic building block to provide *overlay multicast*. Our approach will be to create one DHT per multicast group, and whenever a node requests to multicast information to a group, it broadcasts the information to all the nodes within the DHT that represents the multicast group. We will also show how this scheme can take advantage of IP multicast where it is available, and thus only use the overlay to connect different IP multicast capable networks.

5.1 Related Work

The nodes in a DHT form a *graph* containing a set of *vertices* and a set of *edges* between the vertices. Each node in the DHT represents a vertex in the graph, and each pointer from a node *a* to another node *b*, represents an edge between those vertices. General purpose algorithms for broadcasting to all nodes in such a graph have existed for many decades in the field of distributed algorithms [137, Chapter 4] [94, pg 496ff] [13, Chapter 2]. There is, however, a fundamental difference between those algorithms and the ones we present in this chapter.

Graphs may contain *cycles* (see Figure 5.1). A cycle starts at a vertex and visits other vertices by traversing edges, and ends in the start vertex, without ever visiting any vertex more than once, except for the start vertex, which is visited twice.

The possibility of cycles makes broadcasting in graphs complicated.

Generally, broadcast algorithms for graphs are constructed to avoid messages traveling in cycles indefinitely. These algorithms have two disadvantages. First, they bear an additional message complexity as some messages will be redundant. For example, if the graph consists of three nodes $\{a, b, c\}$, with edges between (a, b), (b, c), and (c, a), only two messages would suffice to broadcast from any node to all other nodes. General broadcast algorithms for graphs would, however, use four messages to cover those nodes, because each node would send the message to all neighbors, except the one it got the message from. For example, if a initiated the broadcast, a would send to b and c, b would send to c, and c would send to b. Second, such algorithms use extra state at each node to avoid messages wandering in cycles indefinitely. In the previous example, each node would keep track of the received messages. Thus, node b would drop the message from node c, and node c would drop the message from node b.

Even though DHTs contain cycles, the algorithms we present in this chapter make use of the internal structure of DHTs to avoid the redundant messages and the extra state associated with avoiding cycles.

5.2 Model of a DHT

Our goal is to construct general group communication algorithms which can be used on as many DHTs as possible. Hence, we try to not depend too much on structure induced by the routing pointers. Our aim is to make our algorithms applicable to any ring-based DHTs. Nevertheless, ring-based DHTs significantly differ in how they place routing pointers. For example, in many DHTs — such as Pastry [123], P-Grid [2], and Tapestry [143] — routing pointers are placed according to prefixes of the identifiers of the nodes. Other DHTs — such as Chord [134], DKS [5], and SkipNet [65] — place pointers according to the relative distance of nodes on the ring. We will disregard such differences and represent the routing pointers uniformly.

Our assumption is that the system is ring-based, i.e. each node has an identifier from an identifier space of a fixed size N, and each node has a pointer to its predecessor and successor on the ring. Furthermore, we represent the routing pointers of each node in monotonically increasing distances, similarly as we did in (see Section 4.4.1). That is, all the

pointers of a node n are represented by a function[1], rt, where $rt(1)$ points to the successor of n, $rt(2)$ points to the node in the routing table of n which is the second closest to n going in clockwise direction starting at n, etcetera. Hence, f sorts all the pointers of a node n according to their distance to n, with $rt(1)$ being the successor of the node, and $rt(K)$ being the predecessor of the node, given that the node n has K pointers. The variable K varies from node to node and from time to time.

We avoid redundant pointers to simplify our algorithms. For example, as seen by Figure 4.1, some pointers of a node might be pointing at the same node, i.e. $rt(3) = rt(4) = 10$. Our algorithms will be simplified if all the pointers are unique. Therefore, we represent the pointers by another function u. The successor of a node is always represented by $u(1)$, and $u(2)$ points to the second closest node etcetera, while guaranteeing that $u(i) \neq u(j)$ for any two distinct i and j. We assume that a node has M unique pointers. Again, M might vary from node to node and from time to time. For convenience, we sometimes use $u(0)$ to denote the identifier of the local node, i.e. $u(0)$ is always a self pointer, but it is not counted by M.

The above function u can be formally defined as follows. Let the set R contain all the pointers of a node n, i.e. $R = \{rt(k) \mid 1 \leq k \leq K\} \cup \{n\}$. We define the *local successor function* at a node as:

$$s(i) = i \oplus min(\{j \ominus i \mid j \in R\})$$

The function u at node n can now be defined for the domain $[0, |R| - 1]$ as :

$$u(i) = \begin{cases} n & \text{if } i = 0 \\ s(u(i-1) \oplus 1) & \text{otherwise} \end{cases} \tag{5.1}$$

We will initially assume that all pointers are correct and no failures occur.

Our algorithms greatly benefit if they are used together with atomic ring maintenance and the accounting algorithms presented in Chapter 4, as they guarantee that all routing pointers are correct unless some node has failed.

[1] Alternatively, the function can be perceived as a data structure, such as an array.

5.3 Desirable Properties

The group communication algorithms we present in this chapter share the following correctness properties:

- *Termination*. Eventually the algorithm should terminate.
- *Coverage*. All the *designated* nodes that are *reachable* from the initiation to the termination of the algorithm should receive the message.
- *Non-redundancy*. No node should ever receive a message more than once.

The two first properties are liveness properties, while the last property is a safety property. The last two properties bear some more discussion.

We discuss the terms *designated* and *reachable*, as it comes to the coverage property.

The particular group communication algorithm determines what constitutes a designated node. For example, in a broadcast algorithm all nodes are designated nodes. In some other algorithms, perhaps only a subset of the nodes are considered, e.g. all nodes with identifiers in a certain interval.

The nodes that can be visited by traversing the successor pointers, starting at some initiator and stopping whenever the initiator is reached or overshot (see Section 4.3), are regarded as nodes reachable from that initiator. We assume that the pointers of any node, as defined by the function u, always point at some node which is reachable from the initiator.

The reason that the coverage property is technically involved is to avoid idiosyncrasies that are theoretically possible in a ring-based DHT. For example, one can construct a *loopy* ring, where the state of the successor and predecessor pointers seems correct from the perspective of any two neighboring nodes, but the overall ring structure is inconsistent. Figure 5.2 shows a loopy ring, where pointers between the successor and a predecessor of a node are symmetric, i.e. for every node u, u's successor has a predecessor pointer pointing at u. However, between any node and its successor there is another node. Consequently, if the ring is traversed starting at some initial node and stopping whenever the initial node is reached or overshot, not all the nodes in the system have been visited. It is not clear how such a loopy network could occur. There are algorithms

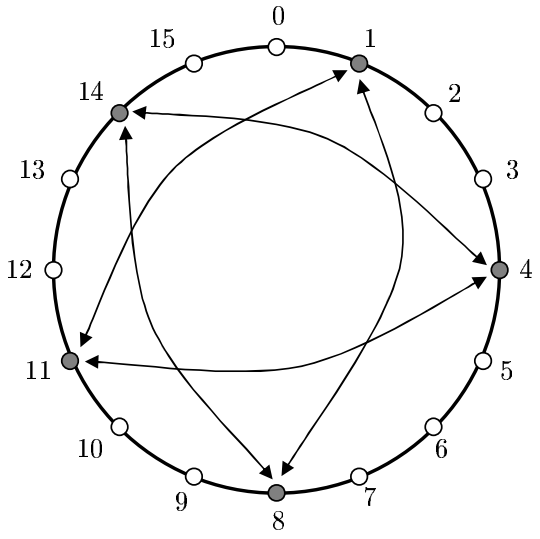

Figure 5.2: A loopy network where $(u.succ).pred = u$ for every node u, but for every node u there is a node v between u and $u.succ$.

for making such ring states consistent [91, 90], but it is not in the scope of our group communication algorithms to deal with such scenarios.

By the last correctness property, non-redundancy, we mean that the same message should never be transmitted more than once to the same node. This is different from non-redundant *delivery* of messages, which is adopted by others [76, pg 33f]. The latter is achieved by associating a globally unique identifier with every invocation of the algorithm, and filtering any message with previously seen identifiers. Thus, the application is delivered the message at most once, even though the node might receive the same message multiple times. This comes at the cost of transmitting redundant messages, and maintaining state associated with every invocation of the algorithm.

5.4 Broadcast Algorithms

In this section we provide algorithms for broadcasting to all nodes in the DHT. Hence, every node is considered to be a designated node by the algorithm.

A naïve broadcast algorithm would start at the initiating node and

traverse the successor pointers, ensuring that all nodes receive the message. The algorithm would terminate whenever the successor pointer of a node points to the initiator or overshoots the initiator. In other words, if p initiates the algorithm, a node q would only forward the request to its successor if $q.succ \in (q, p)$.

The naïve algorithm is correct. The algorithm would eventually terminate in the presence of churn as it is impossible to keep forwarding the message to a successor infinitely without ever using a successor pointer which points at, or overshoots, the initiator. If atomic ring maintenance is used, the algorithm would cover all reachable nodes that remain in the system until the algorithm terminates. No node would ever get the message more than once, as the condition for forwarding ensures that nodes that previously received the message are not forwarded to.

The algorithm would, however, incur a message complexity of $O(n)$ and a time complexity of $O(n)$, where n is the number of nodes in the system.

5.4.1 Simple Broadcast

The naïve algorithm can be improved. Assume the identifier space is $\{0, \cdots, 1023\}$, and only nodes with odd identifiers exists, i.e. 1, 3, 5, $\cdots$, 1023. Assume node 1 is to broadcast a message, and it happens to have node 511 in its routing table. A simple improvement of the naïve algorithm would be to let the initiator 1 *delegate* responsibility to node 511 to traverse the successor pointers in the range 512 to 0 on the ring, while 1 itself traverses the successor pointers in the range 2 to 510. Our message complexity will still be n, but our time complexity will be halved to $\frac{n}{2}$. It is obvious that all nodes in the system would be covered, and that no redundant messages would be sent. Our *simple broadcast algorithm* applies the idea of delegation recursively for every pointer that points to a node that has not previously been sent to.

The simple broadcast algorithm, shown by Algorithm 19, works as follows. The initiating node partitions the identifier space into M parts, and delegates to each routing neighbor the responsibility to cover all nodes in its part. More concretely, the initiating node i delegates responsibility to node $u(M)$ to cover all nodes in the interval $(u(M), i)$, node $u(M-1)$ is delegated responsibility to cover $(u(M-1), u(M))$, etcetera, down to node $u(1)$ (successor of i) to cover nodes in $(u(1), u(2))$. Each delega-

tee further partitions the part to which it has been delegated into pieces which it further delegates to other nodes. This is recursively repeated until no node has any routing pointers left in the interval it has been delegated, whereby the algorithm terminates.

Algorithm 19 Simple broadcast algorithm

```
1: event n.StartSimpleBcast(msg) from app
2:     sendto n.SimpleBcast(msg, n)              ▷ Local message to itself
3: end event

1: event n.SimpleBcast(msg, limit) from m
2:     Deliver(msg)                              ▷ Deliver msg to application
3:     for i := M downto 1 do                    ▷ Node has M unique pointers
4:         if u(i) ∈ (n, limit) then
5:             sendto u(i).SimpleBcast(msg, limit)
6:             limit := u(i)
7:         end if
8:     end for
9: end event
```

We can now prove that the algorithm is correct given a static network.

Theorem 5.4.1. *The simple broadcast algorithm is correct.*

Proof. Termination. *The algorithm only forwards a message to a node $u(i)$ if $u(i)$ is in the interval $(n, limit)$. Hence, the beginning of the interval strictly increases (modulo arithmetic) toward limit each time a message is forwarded. Similarly, the end of the interval (limit), either stays the same or decreases toward n. Hence, each time a node sends a message to some other node, it delegates to it a strict subset of the interval that itself was responsible for. Since the intervals are discrete and have a finite size, eventually there is either no node in the interval or the interval is empty. Hence, eventually the algorithm terminates.*

Non-redundancy. *We have shown that each node delegates a subset of its own interval to any node it forwards to. Hence, a tree is induced by the broadcast, where the initiator is the root of the tree, and where a node is the immediate parent of all the nodes it directly sends a message to. We show that every node delegates non-overlapping intervals to all its children. This is is a consequence of the fact that the pointers at any given node are unique and a node never*

sends a message to itself, since 0 *(and consequently* $u(0)$*) is not covered by the index i in the for-loop. Hence, every node n is delegated responsibility to cover an interval* $(n, limit)$*, and n sends a message to any neighbor in that interval, delegating each of them non-overlapping subsets of* $(n, limit)$*. Furthermore, the interval delegated to a node by a node n never contains the identifier of any of n's children.*

By induction on the broadcast tree, the same node cannot occur more than once in the subtree of any node. The statement is true for the base case, which is any tree containing only one node. For the induction step, assume the hypothesis is true for the subtrees, $T_1, \cdots, T_n$ *that are formed by an arbitrary node p's respective children. Then it is also true for the subtree formed at p, since the intervals of each tree is non-overlapping and p does not occur in any of those intervals. Hence, no node ever receives the same message twice.*

Coverage. *By induction on the broadcast tree, the subtree at a node p covers all the nodes that have an identifier in that interval. It is true for the base case, which is any tree containing only one node p. Since the interval given to p is of the form* $(p, limit)$*, no node in the system can have an identifier in that interval, otherwise one of them would be p's successor* $u(1)$*, contradicting that p is the only node. For the induction step, assume the hypothesis is true for the subtrees of p's respective children. Then all the intervals delegated to p's children have been covered, which only leaves the identifiers of p's children to be covered for p to ensure that its delegated interval is covered. But those identifiers are covered since p directly sends a message to its children. Hence, p and its children will cover all the nodes in the interval delegated to p. Since the initiator starts with the whole identifier space as its delegated interval, all nodes will be covered.*

□

The simple broadcast algorithm is in fact broadcasting over the virtual k-ary tree (see Figure 4.7). Consequently, the time complexity of the algorithm is $\log_k(n)$, for n nodes, given that the pointers are selected according to the k-ary principle described in Section 4.4. Furthermore, the message complexity is n, since all nodes receive the message and no node receives the message more than once. Most importantly, no single node ever needs to send a message to more than M nodes, given that it has pointers to M nodes.

Figure 5.3 shows the example of a ring and Figure 5.4 shows the how a simple broadcast would disseminate over that ring if a broadcast was initiated by node 1.

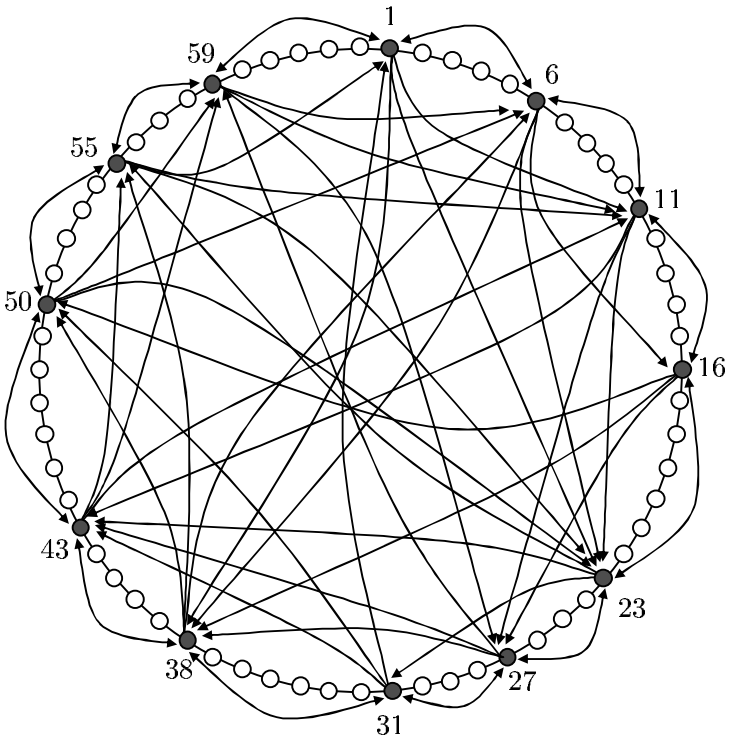

Figure 5.3: The figure shows an identifier space $\{0, \cdots, 63\}$ and 12 nodes with identifiers $1, 6, 11, 16, 23, 27, 31, 38, 43, 50, 55$, and 59. Single arrowed lines represent a routing pointer and each double arrowed line represents a node's predecessor pointer and the corresponding successor pointer.

5.4.2 Simple Broadcast with Feedback

The operation provided by the simple broadcast algorithm is useful for some applications, such as overlay multicast or publish/subscribe systems. It is, however, not sufficient if the broadcasting node wishes to receive a feedback or a response from the nodes it is broadcasting to. This is especially the case if broadcasting is used to implement arbitrary or complex queries. A naïve solution would be to inform all nodes of the identity of the initiator, and let them directly send their feedback to the initiator. But this is not scalable as the initiator quickly becomes a bottleneck as the number of nodes becomes large.

The *Simple Broadcast with Feedback Algorithm* (Algorithm 20) efficiently collects responses from all nodes after broadcasting. It extends the simple

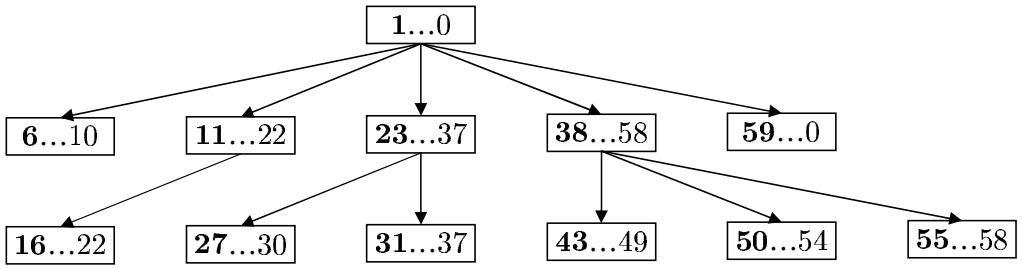

Figure 5.4: The figure shows how a simple broadcast initiated by node 1 would disseminate in the system depicted by Figure 5.3. Each box represents the node receiving the broadcast message (in bold) and the interval that it is delegated responsibility to cover.

broadcast algorithm by letting each node maintain a set, *Ack*, of all nodes that it has sent the message to. Each node also has a variable *par*, pointing to the parent, from which it received the broadcast message. Every node waits to receive a response from each of the nodes in its *Ack* set before it sends its own response together with the gathered responses to *par*. Naturally, nodes that are delegated an interval in which they have no neighbors, can immediately send their response to *par* as they do not need to wait on a response from any node. As a side note, these will be the nodes that are the leafs of the induced broadcast tree.

The feedback algorithm has twice the time and message complexity as the simple broadcast algorithm. If the *k*-ary principle is used for the routing pointers, each node sends maximum $(k-1)\log_k(N)+2$ messages, where $(k-1)\log_k(N)+1$ accounts for all the routing pointers plus the predecessor pointer, and the response back to the parent accounts for another message.

Exploiting Atomic Ring Maintenance The algorithms that collect feedback require that every node stays in the system until it receives a feedback from its children. It is possible to make each node that is involved in the broadcast wait to receive feedback from all its children before it leaves the system. This can be achieved by exploiting the locking scheme described in Chapter 3. Before a node sends a message to a node p, it ensures that it holds the lock L_p. Similarly, a node releases its own lock right after sending the response back to its parent.

Algorithm 20 Simple broadcast with feedback algorithm

```
1: event n.StartBcast(msg) from app
2:     sendto n.Bcast(msg, n)                    ▷ Local message to itself
3: end event

1: event n.Bcast(msg, limit) from m
2:     FB := Deliver(msg)                ▷ Deliver msg and get set of feedback
3:     par := n
4:     Ack := ∅
5:     for i := M downto 1 do                    ▷ Node has M unique pointers
6:         if u(i) ∈ (n, limit) then
7:             sendto u(i).Bcast(msg, limit)
8:             Ack := Ack ∪ {u(i)}
9:             limit := u(i)
10:        end if
11:    end for
12:    if Ack = ∅ then
13:        sendto par.BcastResp(FB)
14:    end if
15: end event

1: event n.BcastResp(F) from m
2:     if m = n then
3:         sendto app.BcastTerm(FB)
4:     else
5:         Ack := Ack − {m}
6:         FB := FB ∪ F
7:         if Ack = ∅ then
8:             sendto par.BcastResp(FB)
9:         end if
10:    end if
11: end event
```

5.5 Bulk Operations

In this section we generalize the problem and provide two operations: *bulk operation* and *bulk owner operation*. Bulk operation takes a set of identifiers I, referred to as the *bulk set*, and sends a message to all nodes that have identifiers in the set I. Hence, the designated nodes are all nodes with an identifier in the bulk set I. Bulk owner operation takes a set of identifiers I and broadcasts to the nodes that are *responsible* for the identifiers in the set I. The notion of responsibility is the same as defined in Chapter 2, where every node n is responsible for all identifiers between its predecessor and itself, i.e. the identifiers $(n.pred, n]$. Hence, the designated nodes are those which are responsible for an identifier in the bulk set I.

Naïve Solution The simple broadcast algorithms presented in the previous section can easily be modified to only broadcast to parts of the ring. The initiating node i can set *limit* to any identifier, and the simple broadcast algorithm will ensure that only nodes in the range $[i, limit)$ get the broadcast message.

It would be desirable if an initiating node i could broadcast to any node in an arbitrary interval $(k, m]$. This could naïvely be achieved if the initiating node first routes an ordinary lookup message to the successor of k, which then broadcasts to everyone in $(k, m]$. There is, however, a more efficient solution which we describe next.

Motivation A main motivation for bulk operation is that it can be used to build a *bulk lookup* or *bulk insert* operation. In many applications, such as file systems built on top of DHTs, it is desirable to lookup many keys in parallel. We have encountered file systems built on top of DHTs, in which thousands of simultaneous lookups were issued to fetch a large file [10, 132]. In such cases, there is a significant overhead induced by marshaling and sending thousands of lookups. With the bulk owner operation, the identifiers of all those keys would be described by the bulk set I, which is then used to do parallel lookups. The advantage of the bulk operation is that it guarantees that a node will send at most as many messages as it has pointers, which is often $O(\log n)$ pointers in an n node system. Furthermore, if the routing pointers are placed according to the k-ary

principle (see Section 4.4), the worst case time complexity of the whole operation is $O(\log n)$, in an n node system. Bulk operation improves the bit complexity when compared to making parallel lookups for every identifier. Hence, it does not trade bit complexity for message complexity.

The bulk operation algorithm has two extreme cases. If the whole identifier space is used as I, it reduces to the simple broadcast described in the previous section. If only one identifier is used as I, the algorithm reduces to a simple lookup. We therefore think that the bulk operation should be the basic operation provided in DHT APIs.

We have found many applications of the bulk operation. We use it in Section 5.6.1 to provide a fault-tolerant broadcast, which uses the bulk operation to retry to cover an interval which was delegated to a node that has failed. We also use it in our symmetric replication scheme, which is described in Chapter 6.

5.5.1 Bulk Operations Algorithm

The bulk operation algorithm is given by Algorithm 21. It is very similar to the simple broadcast algorithm (Algorithm 19). The algorithm is initiated by sending a BULK message with two parameters. The first parameter I is a set of identifiers, while the second parameter msg is the message to be sent to all nodes with identifiers in I.

One major difference between the bulk algorithm and the broadcast algorithm is that in the broadcast algorithm, every node that received a message also delivered it to the application, while in the bulk algorithm some nodes might be acting as forwarders, which never deliver messages to the application.

Another difference with the simple broadcast algorithm is that it only sends a message to a node $u(i)$ if $u(i)$ has an identifier in I, or if there are identifiers in I which are between $u(i)$ and $u(i+1)$, in which case $u(i)$ is the closest preceding node which can forward the request closer to any potential nodes in that range. This is implemented by creating a set of identifiers $J := [u(i), limit)$, and only sending a message to $u(i)$ if the intersection of I and J is non-empty. When sending a message to $u(i)$, only identifiers in the intersection of I and J are delegated to $u(i)$.

Whenever an interval $I \cap J$ is delegated to some node $u(i)$, that interval is removed from I to ensure that no two nodes are delegated overlapping intervals.

Algorithm 21 Bulk operation algorithm

```
1: event n.BULK(I, msg) from m
2:     if n ∈ I then
3:         Deliver(msg)                          ▷ Deliver msg to application
4:     end if
5:     limit := n
6:     for i := M downto 1 do                    ▷ Node has M unique pointers
7:         J := [u(i), limit)
8:         if I ∩ J ≠ ∅ then
9:             sendto u(i).BULK(I ∩ J, msg)
10:            I := I − J                        ▷ Same as I := I − (I ∩ J)
11:            limit := u(i)
12:        end if
13:    end for
14: end event
```

Figure 5.5 shows an example of how the bulk message disseminates in the system depicted by Figure 5.3. Node 1 initiates the algorithm, and wishes to broadcast to all nodes in the interval $[30, 45]$. Note that a bulk message is sent to node 27 with the responsibility of covering the interval $[30, 30]$. This might seem unnecessary as node 27 is only a forwarder that does not deliver the message to the application layer, nor does it forward the message to any other node. Nevertheless, node 23 which delegated the interval $[30, 30]$ to node 27, has to ensure that it covers all nodes in the region $[30, 37]$ and does not know whether a node with identifier 30 exists or not. Therefore it delegates that interval to node 27, which knows that no such node exists.

5.5.2 Bulk Operations with Feedbacks

Algorithm 22 shows the algorithm for doing bulk operation with feedback from all the nodes with identifiers in a prescribed bulk set I.

In the simple broadcast with feedback, the nodes that are merely forwarding the message will not provide any feedback to the initiator. Nevertheless, feedback on its way back to the initiator might have to pass through such forwarders, hence forwarding nodes should be placed in the waiting *Ack* set.

Algorithm 22 Bulk operation with feedback algorithm

```
1:  event n.BulkFeed(I, msg) from m
2:      if n ∈ I then
3:          FB := Deliver(msg)              ▷ Deliver and get set of feedback
4:      else
5:          FB := ∅                                          ▷ No feedback
6:      end if
7:      par := m
8:      Ack := ∅
9:      limit := n
10:     for i := M downto 1 do              ▷ Node has M unique pointers
11:         J := [u(i), limit)
12:         if I ∩ J ≠ ∅ then
13:             sendto u(i).BulkFeed(I ∩ J, msg)
14:             I := I − J                  ▷ Same as I := I − (I ∩ J)
15:             Ack := Ack ∪ {u(i)}
16:             limit := u(i)
17:         end if
18:     end for
19:     if Ack = ∅ then
20:         sendto par.BulkResp(FB)
21:     end if
22: end event

1:  event n.BulkResp(F) from m
2:      if m = n then
3:          sendto app.BulkFeedTerm(FB)
4:      else
5:          Ack := Ack − {m}
6:          FB := FB ∪ F
7:          if Ack = ∅ then
8:              sendto par.BulkResp(FB)
9:          end if
10:     end if
11: end event
```

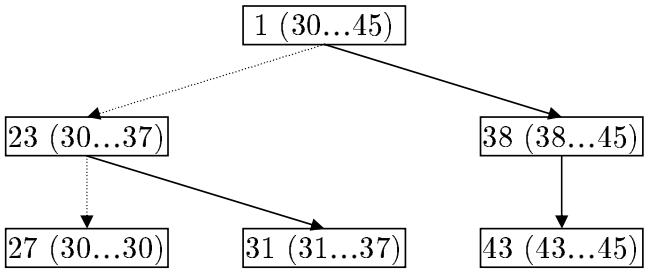

Figure 5.5: The figure shows how a bulk message would disseminate in the system depicted by Figure 5.3. The bulk operation is initiated by node 1, who wishes to send a message to all nodes in the interval $[30, 45]$. Each box represents the node receiving the broadcast message and in parenthesis the interval that it is delegated responsibility to cover. Dotted arrows indicate that the receiving node is merely a forwarder which will not deliver the message to the application.

5.5.3 Bulk Owner Operations

We now extend the bulk operation to introduce the bulk owner operation, which is designed to reach all the nodes that are responsible for an identifier in the bulk set I.

Algorithm 21 reaches every node that has an identifier in the bulk set I. These nodes should also be reached by the bulk owner algorithm, since every node is responsible for its own identifier. Sometimes, however, a node n responsible for an identifier in the bulk set is not itself in the bulk set, i.e. $n \notin I$. We first show how Algorithm 21 can be naïvely changed to accomplish this, and thereafter we optimize it.

The simplest way to ensure that all the designated nodes are reached is to add the statements found in Algorithm 23 after the for loop in Algorithm 21 . Hence, each node n first executes the statements in Algorithm 21 and thereafter checks to see if there are any identifiers in the bulk set I that are between itself and its successor, i.e. $(n, u(1)]$, in which case it sends a message to its successor $u(1)$.

The naïve extension has one major drawback, it might deliver the same message to the same node multiple times. Algorithm 24 optimizes the naïve extension to avoid the sending of redundant messages.

We modify the algorithm to add another parameter R, which holds a set of identifiers. A node sending a BULKOWN message sets R to the

Algorithm 23 Extension to bulk operation

```
1: J := (n, u(1)]
2: if I ∩ J ≠ ∅ then
3:     sendto u(1).Bulk(I ∩ J, msg)
4: end if
```

set of identifiers which the destination node is potentially responsible for. Initially, R is equal to the bulk set I. This set R is always checked when receiving a BulkOwn message to determine if there are any identifiers in R that the local node is responsible for.

The first modification to avoid redundant messages is to let every node keep a boolean variable *sendsucc* which indicates whether the node sent a message to its successor or not. If *sendsucc* is true after executing the for loop, the node will not again send a message to its successor.

The last described modification to the algorithm does not ensure that a node will not send a redundant message. Even if *sendsucc* is false, the successor might have received the BulkOwn message from some parent of the current node. The second modification is to include a parameter *next* whenever sending a message to any node m, where *next* is the closest successor of m which is known to have received the BulkOwn message. Hence, a node will not send a message to its successor if *next* is its successor, or if *sendsucc* is true. Initially, *next* is set to the identifier of the initiator.

5.6 Fault-tolerance

So far we have assumed that failures do not occur. If the goal is to have best effort delivery, then the broadcast and the bulk algorithms will be providing best effort delivery in the presence of failures. However, this is not the case with the algorithms which provide feedback. In those, every node, prior to sending its feedback, waits to receive feedback from all nodes to which it has sent a message. If any of those nodes fail, the waiting node will block forever, making the whole algorithm deadlock.

A straightforward approach to ensure termination for all algorithms, is to introduce timeouts. Hence, a node waiting for feedback, can time out and send its response back to its parent. A premature timeout might

Algorithm 24 Bulk owner operation algorithm

```
1: event n.StartBulkOwn(I, msg) from m
2:     sendto n.BulkOwn(I, I, n, msg)                ▷ Local message to itself
3: end event

1: event n.BulkOwn(I, R, next, msg) from m
2:     MS := R ∩ (u(M), n]                                ▷ u(M) is same as pred
3:     if MS ≠ ∅ then
4:         Deliver(msg, MS)                ▷ App is responsible for ids in MS
5:     end if
6:     limit := n
7:     lnext := next
8:     sentsucc :=false
9:     for i := M downto 1 do                ▷ Node has M unique pointers
10:        J := (u(i), limit]
11:        if I ∩ J ≠ ∅ then
12:            K := (u(i − 1), u(i)]
13:            sendto u(i).BulkOwn(I ∩ J, I ∩ K, lnext, msg)
14:            I := I − J                              ▷ Same as I := I − (I ∩ J)
15:            limit := u(i)
16:            lnext := u(i)
17:            if i = 1 then
18:                sentsucc :=true
19:            end if
20:        end if
21:    end for
22:    J := (n, u(1)]
23:    if I ∩ J ≠ ∅ and sentsucc = false and next ≠ u(1) then
24:        sendto u(1).BulkOwn(∅, I ∩ J, limit, msg)
25:    end if
26: end event
```

result in some nodes sending their feedback to a parent which has timed out. Such messages can simply be ignored by the node which timed out.

A best effort delivery might, however, be inadequate. For example, assume the initiating node attempts to broadcast to all nodes in the system. Also assume that it has $M = \log_2(n)$, pointers placed as in Chord (or according to the k-ary principle when $k = 2$). In such case, the broadcast will partition the whole space into M intervals and delegate responsibility to each routing pointer $u(i)$. The pointer $u(M)$ will be pointing to the predecessor of the initiator, and $u(M-1)$ will be delegated roughly half of the identifier space. If $u(M-1)$ fails, then roughly half of the nodes will not be reached by the broadcast. This is exacerbated as $u(M-1)$ might not have failed, but just left the system, and the initiator is not yet aware of it. Hence, even with reliable communication channels and no failures, the algorithm might fail to cover significant number of nodes. The latter can be avoided if atomic ring maintenance is used together with the accounting algorithms presented in Chapter 4.

In the distributed algorithms field, *reliable broadcast* has been studied extensively [62, 26, 27]. Reliable broadcast ensures two things. First, if the initiator of a broadcast does not fail, all correct nodes eventually receive the message. Second, all correct processes always receive the same set of messages, regardless of any crash failure[2].

The second requirement of reliable broadcast is quite strong. For example, assume an initiating node starts to broadcast a message and then fails before completing. If some nodes receive and deliver the message, then all other correct nodes must also eventually receive that message and deliver it. Such a strong requirement, however, is justified in many scenarios. Reliable broadcast is used as a building block in middleware for building reliable distributed systems, such as Isis [17] and Transis [9]. Reliable broadcast, with the additional constraint that all nodes should deliver messages in the same order, has been shown to be equivalent to the *consensus* problem [26], which is an important theoretical problem with many implications in distributed computing [46].

Our goal is to make broadcast or bulk algorithms, which give stronger guarantees than pure best-effort delivery, but weaker guarantees than reliable broadcast. The motivation for this is that reliable broadcast is quite

[2]It also ensures that a received message actually has been sent, such that messages are not "invented" by the algorithm.

costly to achieve. The algorithm given by Chandra and Toueg [26] has a message complexity of $O(n^2)$ for n nodes. Furthermore, in some application scenarios, the initiating node is using broadcast to collect information. If the initiating node fails, there is no interest in ensuring correct delivery to all nodes.

5.6.1 Pseudo Reliable Broadcast

We will modify our simple broadcast algorithm and introduce *pseudo reliable* broadcast. The algorithms will depend on the initiating node not failing. Hence, it has the correctness properties listed in Section 5.3 with two minor modifications. First, the coverage property assumes the initiator does not fail. Second, the designated nodes are all the correct processes. Informally, this means that a broadcast message will reach all processes that remain in the system between the time the algorithm is initiated and terminated, provided that the initiating node does not fail.

The pseudo reliable algorithms will use failure detectors that use timeouts to detect when a node has failed. We assume that the failure detector is strongly complete, which means that it will eventually detect if a node has crashed. The failure detector might, however, not be accurate, which means that it might give false-negatives, suspecting that a correct, albeit slow, node has crashed. The only consequence of inaccuracy is increased bandwidth consumption because of the redundant messages being sent. We assume that whenever a failure detector suspects a failure, it will immediately trigger the correction of the routing information, ensuring that pointers to crashed nodes are eventually removed.

There can, however, be a complication in the implementation of the failure detector. The time it takes for a broadcast to terminate depends on the number of nodes in the system, which in turn determines the depth of the broadcast tree. Therefore, using a timeout when waiting for an acknowledgment from a child is problematic, as the parent does not know the depth of its subtree, and can therefore not determine the time to wait before triggering a timeout. We therefore assume the implementation of the failure detector is independent of the size of the broadcast. This can be achieved by having the failure detector periodically sending a message to its children and awaiting an acknowledgment.

Pseudo-Reliable Broadcast We first describe a simple way to provide pseudo-reliability and thereafter suggest some improvements.

Every broadcast has a globally unique random identifier associated with it, which is included in every message. Nodes keep track of previously seen identifiers and filter any message which has an identifier previously seen. Hence, redundant messages are filtered.

The algorithm works like the broadcast algorithm with feedback. Hence, the pseudo-reliable algorithm makes use of an *Ack* set, containing the children of the node. In addition to keeping the identities of the children in the *Ack* set, the algorithm keeps track of the interval delegated to each child.

The algorithm is made resilient to failures by using the bulk operation to resume after a failure. Whenever a node suspects that one of its children has failed, it uses the *Ack* set to determine the interval delegated to the failed node. Thereafter, the bulk algorithm is used to cover all nodes in that interval. The bulk algorithm will retry to cover all nodes in that interval. If the suspicion is incorrect, then the receiver will ignore the previously seen message. The algorithm terminates whenever the initiator's *Ack* set is empty. To ensure that the algorithm terminates, it is required that no node inaccurately suspects it children for a long enough time period, such that all *Ack* sets become empty. This requirement is ensured by an eventually perfect failure detector. Note, however, that this failure detector provides a stronger guarantee, as it ensures that, eventually, there will be no inaccurate suspicions by any node.

Improving Pseudo-Reliable Broadcast The pseudo-reliable broadcast algorithm can be improved to reduce bandwidth consumption. The basic idea is to try to avoid the re-sending of redundant messages after failures. We motivate this by an example. Assume all the children of a node q, except one, are done covering their delegated intervals. Hence, the *Ack* set of q contains a single child. If q fails, q's parent p will detect that and reassign the interval delegated to node q to a new node r. Hence, the new node r will retry to cover all of q's children, rather than the remaining one. Even though p's children will filter those redundant messages, r still has to consume resources to send those messages.

To avoid redundant messages, nodes could periodically send an update of their current *Ack* set to their parent. This information could be

piggybacked in the response to the heartbeat sent by the failure detector. Hence, nodes periodically report their current *Ack* set to their parent, which upon receipt of the updated *Ack* set updates its child's delegated interval to the received *Ack* set. If the previous example was using the suggested improvement, node q would send its updated *Ack* set, containing its last remaining child, to its parent p. When q later crashed, node p would only delegate an interval containing the remaining child of p to the new node r.

5.7 Efficient Overlay Multicast

In this section we briefly describe how the algorithms presented in this chapter can be used to provide efficient overlay multicast. Overlay multicast can also be perceived as *topic-based publish/subscribe* [45]. In short, a topic-based publish/subscribe system consists of two actors, *subscribers* and *publishers*. A subscriber can subscribe to different topics of interest. Publishers can publish information about an event and a notification of the event will be sent to all subscribers of that particular topic.

The motivation for doing multicast in an overlay network is that the Internet does not provide world-wide multicast capability. The reason for this is that many of the routers that form the back-bone of the Internet have multicast turned off, or do not support it. Several overlay networks have been built, such as Multicast Backbone (MBONE) [44], but these lack the self-managing properties that structured overlay networks possess. Hence, there are several attempts to provide multicast in structured overlay networks [24, 74, 118].

Our approach has several advantages compared to other structured overlay multicast solutions. First, only nodes involved in a multicast group receive and forward messages sent to that group, which is not the case in some other systems [24, 74]. Second, the multicast algorithms ensure that no redundant messages are ever sent, which is not the case with many other approaches [118, 76]. Finally, the system integrates with the IP multicast provided by the Internet, such that the overlay is just used to reach IP-multicast enabled islands, and thereafter IP multicast is used to efficiently reach all nodes in such an island.

5.7.1 Basic Design

The basic idea is to store information about all multicast groups in a DHT, which we refer to as *TOPDHT*. Every multicast group is represented by an instance of an overlay network. To multicast a message to a particular group, the message is broadcast to all nodes in that overlay network, using the pseudo-reliable broadcast algorithm described earlier. This approach is similar to [118], except our system avoids sending redundant messages, and reaches all n nodes in a multicast group in $O(\log n)$ time steps, instead of $O(n^{\frac{1}{d}})$. Chapter 7 describes how this can be implemented efficiently such that as few connections as possible are kept between the nodes in different groups. Figure 5.6 shows an example of a system with two groups.

5.7.2 Group Management

Group information is stored in the TOPDHT using the group name as a key and group information as a value. Group information consists of contact information for a random subset of the members of the group and additional meta-data about the group. The random subset is kept up to date by the node responsible for the item containing the group. The responsible node periodically contacts the first alive node that it knows in its group information, and asks it for its routing table. It then updates its random subset by keeping a constant number of those nodes, giving preference to those just received. The responsible node deletes any group information when it cannot find any live node in a group.

To avoid a responsible node receiving too many requests for a popular group in the TOPDHT, group information is replicated as described in Chapter 6. Furthermore, group information is cached along the lookup path to further relieve nodes in the TOPDHT from hot-spots.

Joining and Leaving Multicast Groups To join a multicast group, a member of the multicast group is obtained by making a lookup for the groups name in the TOPDHT. After obtaining a random subset of the group members, the first alive node in that subset is contacted to join that group. The last node in a multicast group takes special action by deleting the group information in the TOPDHT.

Group Creation An atomic create operation is provided by the nodes in the TOPDHT to avoid conflict resolution when several nodes try to create a group with the same name. Hence, to create a group, a request is sent to the responsible node, which checks to see if such a group already exists, in which case it reports back with a failure. Otherwise, it creates a group having the creating node as the only node in its group information. Hence, the first node that reaches the responsible node will be successful in creating the desired group.

5.7.3 IP Multicast Integration

We provide a mechanism to integrate overlay multicast with IP multicast to make efficient use of the network resources.

The integration of overlay and IP multicast is done by modifying the node behavior when nodes join a multicast group and when they multicast to a multicast group.

Joining a multicast groups works as follows. Every multicast group is associated with a multicast address, which is stored in the TOPDHT together with its group information. The joining node first queries the TOPDHT to find out about existing members of the DHT and the multicast address to be used for that group. Thereafter, a joining node attempts to discover other nodes in the desired group to which it can directly IP multicast. This is done by sending a discovery message to the group's multicast address. If another node receives this message, it responds with its contact information. To avoid an explosion of concurrent responses, similar techniques as used by the Internet Group Management Protocol (IGMP) can be deployed [35]. After establishing contact with an existing node, the joining node joins the desired multicast group using the *same* identifier in the overlay as the existing node. Hence, in each multicast group, nodes that can directly communicate with IP multicast have the same identifier. Therefore, a routing table entry pointing to the successor of some identifier, can have many *candidate* successors. We therefore extend the routing tables to contain up to a constant number of candidates per routing entry.

Broadcasting to a multicast group goes in two steps. The first step is to use the broadcast algorithm to reach one node for every IP multicast island in the multicast group. This is done by using the previously described broadcast algorithm with a minor modification. Since each rout-

ing pointer might contain more than one candidate, the algorithm has to pick one candidate using some desirable metric. The second step of the broadcast is to let every node that receives the broadcast message to IP multicast the message.

The advantage of letting candidate nodes share the same identifier is that no single node need to represent the whole multicast group. Hence, no single node will be a single-point of failure, nor will there exist a single bottleneck for that group.

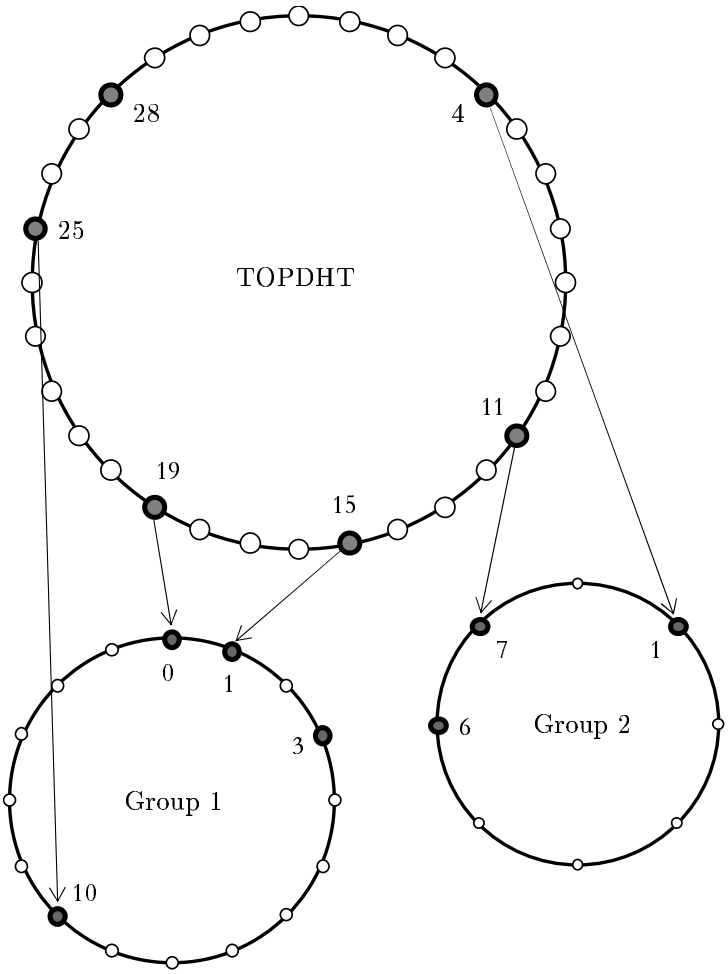

Figure 5.6: A TOPDHT containing information about two different multicast groups. The arrows represent information about random group members in each group.

6 Replication

Chapter 3 and Chapter 4 showed how to form and maintain a structured overlay network, while Chapter 5 showed how such a network could be used for group communication. In this chapter we focus on the DHT abstraction that can be provided by a structured overlay network. The main advantages of DHTs compared to other approaches mainly lie in their ability to self-manage in the presence of node joins, leaves, and failures. Much of this self-management has to do with updating the routing information when nodes join, leave, and fail, as we have shown in earlier chapters. This chapter shows how to self-manage the data stored in the DHT by means of replication, as nodes join, leave, and fail.

6.1 Other Replica Placement Schemes

Most existing DHTs either use *multiple hash functions*, *successor-lists*, or *leaf-sets* for choosing replicas. We shortly describe them and their disadvantages, and thereafter present our proposed replication scheme called *symmetric replication*.

6.1.1 Multiple Hash Functions

Some DHTs — such as CAN [117] and Tapestry [143] — propose using several hash functions for determining the replica placement. In such a scheme, f hash functions are used to achieve a replication degree of f. Each key/value pair in the DHT gets f identifiers by applying the respective functions to the key value.

This scheme, however, has a disadvantage. It requires the inverses of the hash functions to be known to maintain the replication factor. To see why, assume a replication degree of two, and hence two different hash functions, H_1 and H_2, known by all nodes. Assume a node with identifier 10 is storing any items with identifiers in the range $[5, 10]$. Hence, if an item with key "course" gets the identifier $H_1(\text{"course"}) = 7$, it should be stored at the responsible node 10. Assume that 10 fails, and that node 12 becomes responsible for the range $[5, 12]$. Node 12 should then fetch and store the item with key "course" from the other replica to ensure a replication degree of 2. To do this, however, 12 needs to find out the key "course" such that the node responsible for $H_2(\text{"course"})$ can be contacted. Hence, the inverse of the hash function H_1 is required. Even if the inverse of the hash functions were available, each single item that the failed node maintained would be dispersed all over the system when using different hash functions, making it necessary to fetch each item from a different node.

If the replication degree is not restored each time there is a failure, items soon disappear from the system. Assume every node fails with exponential distribution with intensity λ. Then every node fails after an average of $\frac{1}{\lambda}$ time units. Given replication degree f, after an expected $\frac{f}{\lambda}$ time units all replicas of an item would be lost.

6.1.2 Successor Lists and Leaf Sets

Many systems use successor-list replication or leaf-set replication. These two schemes do not suffer from the disadvantages of using multiple hash functions. Successor-list replication [134] works by hashing the key of each key/value pair in the DHT, such that it receives an identifier from the identifier space. Each key/value pair is then stored at the f closest successors of the identifier of the item (see Figure 6.1).

Leaf-set replication [123, 124] is similar to successor-list replication, but rather than storing an item on its closest f successor's, the item is stored on its $\left\lfloor \frac{f}{2} \right\rfloor$ closest successors and its $\left\lfloor \frac{f}{2} \right\rfloor$ closest predecessors. The reason for this difference is that routing always proceeds in clock-wise direction in systems using successor-list replication, while systems using leaf-set replication route in both clockwise and anti-clockwise direction.

These two schemes fulfill two purposes. One purpose is to replicate

items on the successor-list, such that an item stored at a node p is also stored at p's f immediate successors. The advantage of this is that if p fails, lookups can be resolved by its successor, since p's responsibility is automatically shifted to its successor when p fails. The other purpose is to store routing information about f successors, such that as soon as a node p's successor is detected as failed, p's routing information can be updated by replacing the failed node by the failed node's successor. The leaf-set scheme has the same two uses as the successor-list scheme.

Our conjecture is that two separate mechanisms should be used to achieve the above two purposes. While having routing information about the successors or leafs is useful for routing table correction, replication on the same set has several disadvantages.

The first disadvantage is that both schemes need at least f messages for every join and leave event to maintain a replication degree of size f. The reason for this is that if a node leaves the system, its f successors (or its $\left\lfloor \frac{f}{2} \right\rfloor$ predecessors and successors in the leaf-set scheme) will by definition belong to the successor-list (or leaf-set) of a node which they previously were not in. Hence, they need to fetch or release items.

The above scenario can be illustrated by the system shown in Figure 6.1. The figure shows a system with the nodes $0, 2, 3, 5, 6, 8$, and 10 as indicated by the dark circles. Assuming a replication degree of 3, the figure shows that every data item is stored with its three closest successors. Put differently, every node stores the items it is responsible for and replicates all data items stored on its two closest predecessors. Hence, node 5 stores all items in the range $[4, 5]$ and it replicates all items stored on nodes 3 and 2. If node 5 leaves the system or fails, node 6 takes over the responsibility for items in the range $[4, 5]$. Therefore, node 10, which is replicating node 6, needs to be updated with items in the range $[4, 5]$. Furthermore, items stored by node 3 need to be stored on node 8 to restore the replication degree. Similarly, items stored on node 2 need to be stored on node 6 to restore the replication degree. Therefore it is generally required to update f nodes whenever a node leaves or fails, to ensure a replication degree of f.

Furthermore, the re-establishment of the replication degree needs to be coordinated by some node that triggers a replication maintenance algorithm at each of the successors (and predecessors in the leaf-set case).

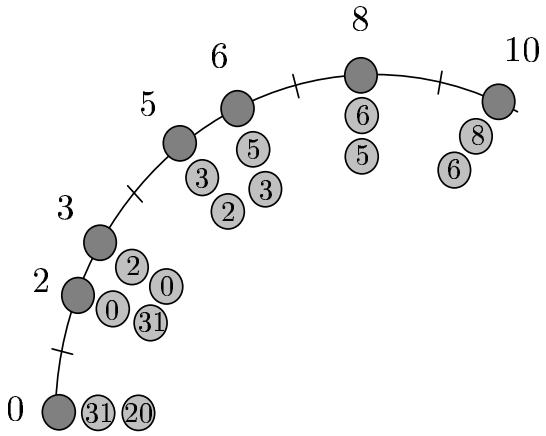

Figure 6.1: A system populated with nodes $0, 2, 3, 5, 6, 8$, and 10 as indicated by the dark circles. Assuming a replication factor of 3, the figure shows that every data item is stored with its three closest successors. Put differently, every node stores the items it is responsible for and it replicates all data items stored on its two closest predecessors, as indicated by light circles. Hence, node 5 stores all items in the range $[4, 5]$ and it replicates all items stored on node 3 and node 2.

The coordinating node might however fail or leave the system, making it necessary to use an algorithm that runs periodically. Many implementations, such as Bamboo [15], use an epidemic algorithm, where each node sends a message to its neighbors whenever it detects a change, leading to f^2 messages for each update or time interval in the case of a periodic algorithm, given a replication degree of size f.

Moreover, any request to a specific replica, m, must first be routed to a node in the successor-list, or the leaf set, before it can be forwarded to m. The reason behind this is that the requesting node has no information about the logical identifier of the replicas, while the nodes in the successor-list, or the leaf-set, maintain such information. In the successor-list scheme, the first replica routed to will always be the clockwise closest replica in the successor-list, while in the leaf-set this can be any of the replicas. In both systems, however, the first replica met is a bottleneck, which can fail, decelerate the whole operation, or in the case of an adversary, launch a malicious attack.

The leaf-set scheme is, however, better in this respect as it naturally

balances requests to different replicas. The reason for this is that it is likely that a request to an item will reach one of its replicas in the last hop before reaching the destination.

6.2 The Symmetric Replication Scheme

Symmetric replication is a general replica placement scheme that can be implemented on top of any DHT. Before presenting symmetric replication, we discuss its benefits.

6.2.1 Benefits

Symmetric replication enables an application to make parallel lookups to exactly k replicas of an item, where $k \leq f$ if the replication degree is f. This has several advantages. A node can use parallel lookups to speed up the lookup process by picking the first response that arrives. It can be used to enhance security by ensuring that the majority of the results match. It is particularly useful if used in conjunction with erasure codes [141], as a random subset of size k of the f replicas can be fetched in parallel to reconstruct the original data.

Another advantage of symmetric replication is that a join or a leave only requires the joining or leaving node to exchange data with its successor prior to joining or leaving. No other exchange of data items is required to restore the replication degree. When compared to successor-list or leaf-set replication, this reduces the message complexity of the restoration from $O(f)$ to $O(1)$, for a replication degree of f. The bit complexity does, however, not change, as the same amount of data needs to be transmitted. But the coordination becomes much simpler and the time complexity improves.

Hence, symmetric replication shares the advantages of using multiple hash functions, but does not suffer from its drawback with restoring replication degrees.

6.2.2 Replica Placement

The main idea behind symmetric replication is that each identifier in the system should be *associated* with f other identifiers. If identifier i is asso-

ciated with identifier j, then the node responsible for item i should store both items i and j. Similarly, the node responsible for item j should store both items i and j.

Formally, each identifier in the system is associated with a set of f distinct identifiers such that the following always holds: if the identifier i is associated with the set of identifiers $r_1, ..., r_f$, then the identifier r_x, for $1 \leq x \leq f$, is associated with the identifiers $r_1, ..., r_f$ as well.

Put differently, the identifier space is partitioned into $\frac{N}{f}$ equivalence classes such that identifiers in an equivalence class are all associated with each other. Any such partition will work, but for simplicity we use the congruence classes modulo m, where N is the size of the identifier space and $m = \frac{N}{f}$ for f replicas.

We now explain how each identifier i is associated to f other identifiers to achieve replication degree f. Let $\mathcal{F} = \{1, ..., f\}$, then identifier i is associated to the f different identifiers given by the function $r : \mathcal{I} \times \mathcal{F} \rightarrow \mathcal{I}$ defined as:

$$r(i, x) = i \oplus (x - 1)\frac{N}{f}$$

Figure 6.2 shows how identifiers are associated in an identifier space of size $N = 16$ and a replication factor $f = 2$. Hence, identifiers form equivalence classes modulo $m = 8$, i.e., identifier 1 and 9 are in the same equivalence class since $1 \equiv 9 (\text{mod } 8)$. The identifiers in the circles represent $r(i, 1)$, while the identifiers outside the circles represent $r(i, 2)$. Note that the association of identifiers is independent from the nodes present in the system.

Nodes replicate data as follows. In a system without any replication, each item with identifier i is stored at the responsible node, which we take to be the successor of item i, but other definitions of responsibility will work as well[1]. Symmetric replication is achieved by having the responsible node of every identifier i storing every item with an identifier associated with i. Hence, to find an item with identifier i, a request can be made for any of the identifiers associated with i.

For example, if the identifier 0 is associated with the identifiers 0, 4, 8, and 12, any node responsible for any of the items 0, 4, 8, or 12 has to

[1] An item with identifier i can be stored at the closest predecessor of i, or at whichever node is closest in the identifier space.

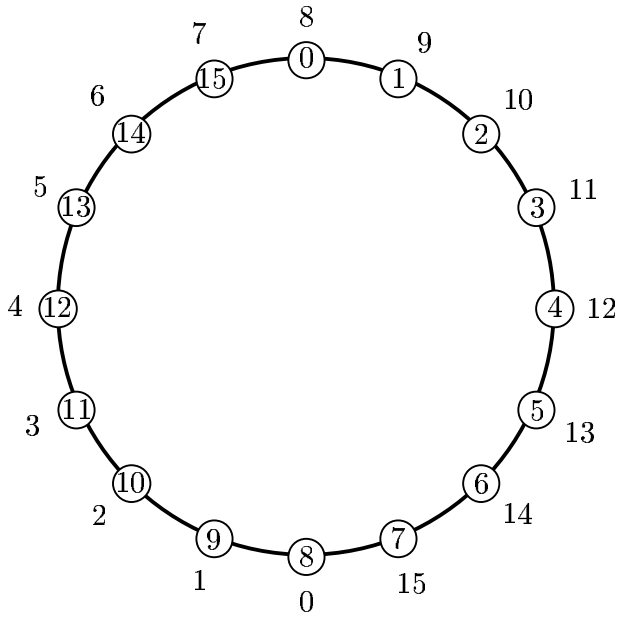

Figure 6.2: The identifiers associated with each identifier in a system with an identifier space of size $N = 16$ and a replication factor of $f = 2$. The identifiers in the circles represent $r(i, 1)$ while the identifiers outside the circles represent $r(i, 2)$.

store all of the items 0, 4, 8, and 12. Hence, to retrieve item 0, a query can be sent to any of the nodes responsible for the items 0, 4, 8, and 12.

For the symmetry requirement to always be true, it is required that the replication factor f divides the size of the identifier space N. We find this reasonable as the size of the successor-list, as well as N, are constants in most systems.

6.2.3 Algorithms

We now give a description of all algorithms. The algorithms will need to be slightly modified to fit a system with a different definition of responsibility, but we assume that each item with identifier i is stored at the successor of i.

Each node in the system has all its items stored in a two-dimensional (f, N)-array denoted *localHashTable*. The first dimension of the array rep-

resents the f identifiers associated with the identifier in the second dimension of the array. Hence, *localHashTable*[i][j] represents items with identifiers $r(j,i)$. In reality, a different representation might be used, to optimize the performance of the local operations.

Join and Leave Algorithms

Whenever a new node n joins the system, it triggers the event JoinReplication (see Algorithm 25) which immediately sends a RetrieveItems message to its successor asking it about all items n should be storing. The message includes information about the items n is interested in by specifying a range (*pred*,*n*], where *pred* is its predecessor's identifier and n is its own identifier.

Once the successor receives the RetrieveItems message, it initializes an empty two-dimensional (f, N)-array called *items*. Thereafter, each item associated with an identifier in the specified interval is copied from *localHashTable* to *items* and sent back in a Replicate message to the newly joined node. Upon receipt of the Replicate message, the newly joined node copies *items* to its *localHashTable*. The new node is now ready to receive lookup requests from other nodes in the system.

The leave algorithm (see Algorithm 25) works similarly to the join algorithm. Whenever a node wants to leave the system it triggers the event LeaveReplication, which uses the RetrieveItems event to copy all items it is responsible for and send them in a Replicate message to its successor. Notice that we do not delete items that are no longer a node's responsibility, though such an operation can be added to avoid a long-running node exhausting its storage space.

Lookup and Item Insertion

Algorithm 26 shows the algorithms used to insert or lookup an item. The algorithms make use of the lookup algorithms described in Chapter 4.

To insert an item, the inserting node simply makes parallel insertions to every location where the replica should be stored. The bulk owner operation (without feedback) can be used to insert an item to all replicas. This has the advantage that both the bit and message complexity can improve, as the same data does not need to travel through the same nodes. Algorithm 26 does, however, not make use the bulk operation.

Algorithm 25 Symmetric replication for joins and leaves

```
1: event n.JoinReplication() from m
2:     sendto succ.RetrieveItems(pred, n, n)
3: end event

4: event n.LeaveReplication() from m
5:     sendto n.RetrieveItems(pred, n, succ)
6: end event

7: event n.RetrieveItems(start, end, p) from m
8:     for r := 1 to f do
9:         items[r] := ∅
10:        i := start
11:        while i ≠ end do
12:            i := i ⊕ 1
13:            items[r][i] := localHashTable[r][i]
14:        end while
15:    end for
16:    sendto p.Replicate(items, start, end)
17: end event

18: event n.Replicate(items, start, end) from m
19:    for r := 1 to f do
20:        i := start
21:        while i ≠ end do
22:            i := i ⊕ 1
23:            localHashTable[r][i] := items[r][i]
24:        end while
25:    end for
26: end event
```

Algorithm 26 Lookup and item insertion for symmetric replication

```
1: event n.InsertItem(key, value) from app
2:     for r := 1 to f do
3:         replicaKey := key ⊕ (r − 1)N/f
4:         n.Lookup(replicaKey,AddItem(replicaKey, value, r))
5:     end for
6: end event

7: procedure n.AddItem(key, value, r)
8:     localHashTable[key][r] := value
9: end procedure

10: event n.LookupItem(key, r) from app
11:     replicaKey := key ⊕ (r − 1)N/f
12:     Lookup(replicaKey,GetItem(replicaKey, r))
13: end event

14: procedure n.GetItem(key, r)
15:     return localHashTable[r][key]
16: end procedure
```

For the lookup algorithm, we only show an event that takes the two parameters *key* and i ($1 \leq i \leq f$) and finds the responsible node for the i:th replica of identifier *key*. On top of this abstraction, different types of lookup services can be built, such as the ones mentioned in Section 6.3.

Handling Failures

Algorithm 27 shows how failures are handled. We assume that the nodes in the network use a failure detector that eventually detects if the successor of a node fails. Inaccuracy, i.e. the detector suspecting that the successor has failed even though it has not, will result in the successor of the suspected node replicating items redundantly. Hence, inaccuracy merely results in inefficiency.

The event FAILUREREPLICATION is triggered at the predecessor of the failed node with parameters specifying the failed node's identifier, the failed node's predecessor's identifier, and an integer specifying which of the f replicas to fetch. Should the restoration of the replicas fail, the process can be repeated by retrying to fetch the replicas from another responsible node.

The failure restoration makes use of the Bulk Owner algorithm (see Chapter 5). Note that the replicas of items stored on the failed node could be dispersed onto several nodes. On average, however, one node will be responsible for the replicas of the items stored on the failed node, as the nodes are uniformly distributed on the ring.

Algorithm 27 Failure handling in symmetric replication

1: **event** n.FAILUREREPLICATION($failed$, *predFailed*, r) **from** m
2: $s := predFailed \oplus (r-1)\frac{N}{f}$
3: $e := failed \oplus (r-1)\frac{N}{f}$
4: **sendto** n.STARTBULKOWN($(s, e]$, RETRIEVEITEMS($s, e, succ$))
5: **end event**

6.3 Exploiting Symmetric Replication

In this section we discuss simple end-to-end techniques that exploit symmetric replication's ability to do parallel requests to replicas to enhance the security and performance of the system.

Distributed voting can be used to ensure that data items received are not tampered with. This is done by sending requests to m replicas and deciding which replica to accept based on a majority vote. The probability that an item has been tampered with can be calculated and reported to the requesting user or application. If the probability that an item is tampered with is p, and m ($2 \leq m \leq f$) parallel requests are made out of which a majority of g ($0 \leq g \leq m$) answers are identical, the probability of such a configuration is given by the Bernoulli trials: $\binom{m}{g} p^g (1-p)^{m-g}$. The system can automatically increase the number of parallel requests m to achieve a certain degree of certainty in the results.

The advantage of symmetric replication is not only restricted to enhancing the security of the system. Symmetric replication can be used to send out multiple parallel requests and picking the first response that arrives. The advantages of this are twofold. First, it enhances performance. Second, it provides fault-tolerance in an end-to-end fashion since the failure of a node along the path of one request does not require repeating the request as it is likely that another one of the parallel requests succeeds. If such a scheme is not used, outgoing messages have to be buffered at a node together with timers, and whenever a timeout occurs, the messages need to be sent again with risk of ending up at the same failed node.

7 Implementation

This chapter briefly describes a middleware called *Distributed k-ary System*, which implements many of the algorithms described in this dissertation. The goal of the chapter is not to describe the architecture of the middleware in detail, but to highlight those parts which we believe are of public interest.

7.1 DHT as an Abstract Data Type

In this section we overview two abstractions that facilitate the usage of DHTs in applications.

7.1.1 A Simple DHT Abstraction

The interface to use a distributed hash table need not be complicated. To this end, we developed *JDHT*, which provides a DHT in the popular programming language Java. The goal of JDHT is to provide an abstraction which has the same interface as an ordinary hash table[1]. Hence, JDHT implements the `java.util.Map` interface and can therefore be used similarly to any other Java map. Thus, JDHT can associate any Java `java.lang.Object` to another `java.lang.Object`. It uses the first object's hash value (obtained with `hashCode()`) as a key in the DHT, and stores it with the second object's serialized representation. Hence, using JDHT locally on one machine is identical to using an ordinary map.

JDHT provides a few additional methods to enable distribution. Every JDHT instance provides a `getReference()` method, which returns a stringified reference to that particular instance of JDHT. This stringified

[1] Also known as a map, a dictionary, or an associative array.

Listing 7.1: JDHT Example

```
JDHT myDHT1 = new JDHT();                    // First node
myDHT1.put("secret","Hello World!");
String ref = myDHT1.getReference();

JDHT myDHT2 = new JDHT(ref);                 // Second node
String helloString = (String) myDHT2.get("secret");
System.out.println(helloString);
```

reference can be supplied as a parameter when creating a new instance of a JDHT, in which case the new instance will attempt to connect the new JDHT node to the overlay network of JDHTs represented by the reference. Listing 7.1 shows an example of two nodes forming a DHT.

7.1.2 One Overlay With Many DHTs

Most applications that use a DHT need to store more than one type of information in the DHT. For example, MyriadStore [132], which is a distributed backup system, uses the DHT for the following purposes. A mapping between user names and current address of nodes is stored in the DHT, and used to enable location of users which have changed network address or location. A mapping between identifiers and contents of directories is used to store metadata about directories. Another mapping between users and their preferences is used to save ordinary application preferences, since a user might want to retain her preferences after her computer has crashed.

Each data type that is stored in the DHT might have different requirements. For example, one might require that the DHT abstraction associates each key to a set of values, such as the group-to-members association given in Section 5.7. Another abstraction might need to associate each key to a single value, such that any put operation overwrites any old value associated with the provided key. This is the case with MyriadStore's mapping of names to network addresses. Other requirements might relate to whether the data in the DHT should be stored on stable storage or which replication degree to use.

In DKS, the application programmer can create many different instances of a DHT and assign them to the same overlay network. Hence,

Listing 7.2: Single Overlay with Multiple DHTs

```
DHT meta = new DHT(dks,"se.kth.mstore.meta",3)
DHT loc = new SingletonDHT(dks,"se.kth.mstore.loc",1)

meta.put("bob",bin_data);
loc.put("bob",ip_address);

ip = loc.get("bob");    // doesn't return bin_data
```

different data types with different requirements can co-exist in the same overlay network. Thus, only one port and one node identifier is consumed per application or machine.

Listing 7.2 shows an example in which two different DHTs are connected to the same overlay network. The first DHT has replication degree 3 and maps each key to a set of values. The second DHT has replication degree 1 and maps a key to a single value. Each instance is given a canonical name. We use a hierarchical name space to avoid name collisions. Both DHTs are connected to the same node in the overlay network, through the object called *dks*. A get operation on a DHT instance only returns those items that have been put into that particular DHT.

The implementation of the mentioned feature is straightforward. Every DHT instance stores with it its canonical name. Any put or get operation carries with it the canonical name of the DHT from which it was issued. Whenever a message arrives at a node, DKS de-multiplexes the message to the right DHT instance using the canonical name as a destination identifier.

Application developers can extend the DHT abstraction by making their own implementation that is tailored to their own needs. For example, a DHT abstraction can be built that stores everything into an external database. As long as every application uses the canonical names consistently, each DHT instance will behave as if it was connected to an independent overlay network.

7.2 Communication Layer

The communication layer provides simple event-based messaging. It consists of the following modules:

- I/O handling module
- Failure detector module
- Multiplexer module
- Marshaling module

The I/O handlers are responsible for buffering, sending, and receiving messages. The failure detector sends heartbeats, awaits acknowledgments, and calculates timeout values that adapt to the latency in the network. The marshaler takes care of unflattening binary data into messages and vice versa. The multiplexer provides an interface, which objects use to dynamically register for events that they are interested in. Hence, the multiplexer dispatches incoming events to the right object.

The rest of this section highlights a few of the properties of the communication layer.

7.2.1 Virtual Nodes

It can be useful for a single machine to join an overlay with multiple identities. This has been suggested for load-balancing purposes, where nodes with more resources can assume several identities to relief other nodes [115, 55]. It has also been suggested as a mechanism to eliminate the natural imbalance that results from the randomness of node identifiers, which makes some nodes responsible for more identifiers than others. Hence, it is avoided that some nodes get to store more items and receive more routing requests than others. By making every node pick $O(\log n)$ identifiers, for an n node network, the imbalance becomes negligible.

DKS facilitates the use of multiple identifiers by providing a single communication manager, on top of which any number of virtual nodes can be registered. Listing 7.3 gives an example of this, where two nodes with identifiers $ID1$ and $ID2$ join the same overlay through the same communication manager cm.

Listing 7.3: Multiple Nodes

```
ComManager cm = new ComManager(2143); // port 2143

DKSNode node1 = new DKSNode(cm, ID1)  // first node
DKSNode node2 = new DKSNode(cm, ID2, node1.getRef())
```

There are several advantages to this design. First, only one IP/port address is consumed per communication manager, regardless of the number of virtual nodes. Second, every node will have its own routing table, but at most one connection is open between any pair of machines. This is particularly useful for some load-balancing schemes, where the routing entries of the virtual nodes on one machine are mostly overlapping [55]. Finally, communication between virtual nodes on the same machine does not have to go through the network. Instead, messages between two local nodes p and q only requires that the multiplexer puts the message from p into q's incoming queue. Hence, the burden of marshaling/unmarshaling and sending and receiving through the OS is completely avoided, making local communication efficient. The same is true for messages from a virtual node to itself, which simplifies the implementation of some algorithms.

The efficiency of local communication greatly simplifies the construction of structured-overlay simulators. The simulator creates a single communication manager, and connects all nodes to this single instance. The simulator handles the scheduling of events, such as joins, leaves, and failures. But any join, leave, or failure, simply means registering or deleting a virtual node to the multiplexer of the communication manager, or deleting a virtual node object without unregistering it from the multiplexer. To enable the simulation of asynchronous networks and latencies, the multiplexer can schedule when to deliver local messages into the incoming queues of the virtual nodes.

7.2.2 Modularity

The communication layer of DKS is modular and can hence be extended for various purposes. We explain two such modules that we have provided different implementations for.

Marshaling Module The marshaling module, is responsible for flattening and unflattening messages sent between the nodes of the distributed system. It provides an interface, where each data type is represented by two methods: one for flattening and one for unflattening. This interface can be used to implement any desirable transport format. Initially DKS provided only an XML based wire format. While this format is great for inter-operability with other systems, it consumes much resources to parse the XML documents passed between the nodes. Therefore, we provide a binary format, which is more compact.

I/O Module DKS provides two implementations of I/O handlers: blocking and non-blocking handlers. The blocking handlers essentially require two threads per connection, one thread listening for incoming traffic, and one thread sending outgoing traffic. The non-blocking handlers are straight-forward finite-state machine translations of the blocking handlers. A thread pool is used together with two finite state machines per connection. Consequently, the non-blocking version can use a constant number of threads regardless of the number of open connections.

8 Conclusion

This dissertation has focused on four topics, each one being the result of the work done on the DKS middleware: lookup consistency, group communication, bulk operations, and replication. As custom, we will review these results here. However, to avoid a monotone description of the results, we will also try to describe the real motivations that lead us to studying these problems.

Lookup Consistency Even though we earlier had worked on the problem of providing lookup consistency, we became seriously aware of the problems during a joint project at SICS. DKS was being coupled with a decentralized authorization server called Delegent, which was storing digital certificates and access policies into the DHT provided by DKS. Some developers noticed strange behavior, when nodes were joining and leaving, some lookups would temporarily report inconsistent results, depending on where they were issued. This motivated us to look into the issue of lookup consistency, as nodes were joining and leaving the overlay network.

Our solution to this problem was divided into two steps. First, we proposed a locking mechanism, similar to the one used in the dining philosophers' problem [37], that would ensure that two neighboring nodes on a DHT ring would never be joining and/or leaving concurrently. Second, we introduced the notion of a *join point* and a *leave point*, which denoted the atomic join, respective atomic leave, of a node. Provided the locking scheme, we showed algorithms that would guarantee that all lookups reported results that were consistent with the join and leave point of the system. The first such solution was based on lock queues, which had some efficiency problems. Therefore, we provided a second solution which was

probabilistic.

We showed how atomic ring maintenance could be augmented to handle arbitrary additional routing pointers. Accounting algorithms were presented that ensure that routing failures never occur as nodes join and leave the system.

The atomic ring maintenance was also considered in the context of node failures. We showed that it is impossible to provide lookup consistency in an asynchronous network that can partition. Hence, we showed that Brewer's conjecture [52] applies to lookup consistency. Our lookup consistency guarantees can therefore be violated during failures. In spite of this, we showed how the algorithms could be made fault-tolerant, by showing how they could be extended and coupled with periodic stabilization. Hence, in absence of failures, the algorithms provide lookup consistency. If failures occur, inconsistent lookup results may be returned. It is left to periodic stabilization to correct the pointers, after which lookup consistency can be guaranteed again.

The presented work advances the state of the art on lookup consistency. Li, Misra, and Plaxton [89, 88, 87] independently discovered a similar approach to ours. An advantage of their work is that they use assertional reasoning to prove safety properties of their atomic ring maintenance algorithms. Their focus has, however, mostly been on the theoretical aspects of this problem. Hence, they assume a fault-free environment. They do not use their algorithms to provide lookup consistency. Furthermore, they cannot guarantee liveness, as their algorithms are not starvation-free. Lynch, Malkhi, and Ratajczak [95] proposed for the first time to provide atomic access to data in a DHT. They provide an algorithm in the appendix of the paper for achieving this, but give no proof of its correctness. As Li *et al.* point out, Lynch *et al.*'s algorithm does not work for both joins and leaves, and a message may be sent to a process that has already left the network [89].

Group Communication Work on broadcast algorithms for structured overlays started already with the publication of El-Ansary *et al.* [42]. The provided algorithm, however, only worked for static networks with perfect routing information. The author joined, and helped with the development of algorithms that could handle incorrect routing entries [49]. This became more relevant when we started using the broadcast algorithms to

build overlay multicast systems [6].

The algorithms in our earlier publications [42, 49, 6, 50] are, however, unnecessarily complex. The reason for that is that they assume that the routing pointers are arranged according to the k-ary scheme. By rearranging the pointers into monotonically increasing distances, and removing duplicate pointers, the algorithms turn into the simple form that is presented in Chapter 5. All algorithms have in common that they guarantee that they reach all nodes within $O(\log n)$ time steps, using $O(n)$ messages, in a system with n nodes. Hence, the overlay multicast system can reach all members of a multicast group in $O(\log m)$ time, using $O(m)$ messages, where m is the size of the multicast group. In contrast to other schemes [24, 74], only nodes involved in a multicast group receive and forward messages sent to that group. Furthermore, the multicast algorithms ensure that no redundant messages are ever sent, which is not the case in some systems [118]. The algorithms are used to provide an overlay multicast system, which efficiently integrates with underlying IP multicast.

Bulk Operations The author has been involved in the design of several file-systems, which are built on-top of DKS [10, 71, 132]. While some of these systems were being built, we faced the problem that the fetching of a single file could sometimes require thousands of lookups to the DHT. Though many of these lookups could be done in parallel, the requesting node still needed to marshal and send thousands of requests. This problem led us to seek algorithms, that would allow us solve problems of this sort.

The *bulk operation* algorithms, which were presented in Chapter 5, enable a node to efficiently make multiple lookups or send a message to all nodes with identifiers in a specified set. The algorithm reaches all specified nodes in $O(\log n)$ time steps and it sends maximum $O(\log n)$ messages per node, where n is the size of the system, regardless of the input size of the bulk operation. This solved our initial problem, where a node needs thousands of simultaneous lookups. The algorithms also proved to be useful when making range queries to all nodes in a certain interval. The bulk operation algorithm also led us to construct a pseudo-reliable broadcast algorithm which repeatedly uses the bulk operation to reach parts of the identifier space that were delegated to failed nodes.

The algorithms also proved useful when doing replication, as described in Chapter 6, and when doing topology maintenance[50].

Replication DKS initially did replication on the successor-list, similarly to many other systems [134, 123]. When implementing the algorithms, however, we found the problem described in Section 6.1. The problem is that every join and leave requires moving items between at least $O(f)$ nodes, where f is the replication degree. To solve it, we had to resort to algorithms which required a message complexity of $O(f^2)$. We found this particularly troublesome, when the size of the items were large. This led us to the symmetric replication scheme, described in Chapter 6, which only requires $O(1)$ messages for every join and leave.

The symmetric replication scheme has other advantages as well. It makes it possible to do recursive parallel lookups, which have been shown to be more resilient to latency variations in the network [120]. Previously, however, iterative lookups have been used to achieve parallel lookups [120, 101], which are known to be costly [120].

8.1 Future Work

We believe that much future work remains on the topics embarked in this dissertation. This includes short-term, as well as long-term research. We start with the short-term research.

Lookup Consistency We believe that it would be interesting to have a formal correctness proof of eventual consistency when atomic ring maintenance is used together with periodic stabilization. We think that this requires a better understanding of periodic stabilization. Periodic stabilization is a non-terminating algorithm that is supposed to run forever. We therefore think that it can be reworked as a self-stabilizing algorithm [38], which always ensures *closure* and *convergence*. Hence, one would prove that the algorithm always converges to a legitimate state, regardless of the starting state, and remains in a legitimate state. By a legitimate state we mean a state in which lookup consistency is satisfied. Such a self-stabilizing algorithm would then always recover from any illegitimate state produced by failures.

Group Communication The efficiency of the group communication algorithms has been calculated assuming that pointers are placed according to the k-ary principle. We believe that it would be interesting to experimentally evaluate the group communication algorithms using other pointer placement schemes. In particular, it would be interesting to evaluate the efficiency of the group communication algorithms if pointers are placed according to the PRR scheme (see Chapter 1). The coverage proof given for the group communication algorithm considers a static network. It would be interesting to see a proof of coverage in the dynamic case.

Strong Replication Consistency

We present some preliminary ideas for providing strong replication consistency guarantees.

It is desirable that a system can give some guarantees on the consistency of the replicated items. For example, assume that some node p updates the value associated with key k to v_1. Shortly, thereafter, some other node q updates key k to v_2. In an asynchronous network, it might be that p's update reaches some replicas of k before q's request, while some other replicas get q's update before p's update. Hence, a lookup to one of the replicas might return either v_1 or v_2. Even if some node makes a lookup to all replicas, it will not be able to know which of the two values is the most recent one, given that no additional information is available. While this might not matter in some applications, other applications might need some consistency guarantees.

A DHT provides a distributed *shared memory* abstraction to applications, where nodes can put and get values to a common shared memory. Hence, it makes sense to adopt the consistency models used in the context of shared memory systems. In the shared memory model, each key is referred to as a *register*. We assume that a put for a key/value pair $\langle k, v \rangle$ simply associates the key k with value v. In the shared memory model, a put is called a *write* and a get is called a *read*.

We now make our discussion about consistency more precise. A node reads or writes a value by issuing a *request*, and thereafter awaits a *response*. In the case of a read response, the value read is returned. In the case of a write response, the requesting node just receives an acknowledgment. We further assume that each request and response is sent at an instant in global time. We say that two operations are not overlapping

if the response to one of the operations arrives before the request of the other operation is made. A weak form of consistency, defined by Lamport [80] is provided by a *regular register*. This consistency model ensures that if there are no operations that overlap in time, any read operation will return the last value written.

As stated earlier, our purpose is to build a system which functions in an asynchronous network with crash failures, such as the Internet. Hence, it is natural to aim at providing replication consistency in the presence of crash failures and network partitions.

It is, however, impossible to implement a DHT which provides regular register consistency in an asynchronous network with network partitions. The result is known as *Brewer's Conjecture* [19] and also relates to the impossibility of lookup consistency, which we provided in Chapter 3.

The conjecture has been formalized and proven by Gilbert and Lynch [52]. We briefly describe their result, which we have reformulated in terms of shared memory registers. The conjecture assumes that the shared memory provides availability and partition-tolerance (see Section 3.5 for a definition) [1].

Theorem 8.1.1 (from [52])**.** *It is impossible in the asynchronous network model to implement a shared memory regular register that guarantees:*

- *Availability*
- *Partition tolerance*

The proof by Gilbert and Lynch is by contradiction. The intuition behind it is that if the network partitions into two components C_1 and C_2, it still needs to provide availability. Hence, any write to a register k in C_1 should eventually terminate. Assume that a non-overlapping read to k in C_2 is requested after the write in C_1 terminated. Also this read should eventually provide a result. Since the network is partitioned, the read in C_2 cannot return the value of the write in C_1. But network asynchrony (see Section 2.1) allows for an identical execution, in which there is no network partitioning, where all messages between the components C_1 and C_2 are delayed until after all the mentioned operations are done.

[1] Gilbert and Lynch model a partition as a network which is allowed to lose arbitrarily many messages sent from one node to another. Hence, a network partition means that messages from the nodes in one component to another are dropped.

This execution is identical to the one in which the network partitioned. Hence, the results of the operations should be the same. But the read in C_2 does not overlap with the write in C_1, yet the read does not return the value of the last write to the register. Hence, regular register consistency is violated.

Circumventing the Impossibility The most common way to circumvent the above problem is to assume that the read and write algorithms can communicate with a majority of the replicas. In a scenario where the network partitions into two components, a majority can only be accessed in one of the components. Hence, availability will be violated in one of the components. Note that it might be impossible to get a majority in any component if the network partitions into more than two components. Such algorithms rely on the fact that any two operations to a majority of the nodes overlap on at least one node. With this assumption regular register consistency, and stronger consistency models, can be implemented.

Getting a majority in a DHT can, however, be problematic. The problem has to do with lookup inconsistency: more than one node might believe it is responsible for a given identifier. Hence, the algorithm assumes there are f replicas, and gets a majority of $\left\lceil \frac{f+1}{2} \right\rceil$, but the number of replicas has actually increased to more than f. Hence, there is no guarantee that two majorities overlap.

The following example illustrates how the number of replicas can increase due to the inaccuracy of the failure detectors. Assume the system consists of the nodes 10, 30, 50, 60, and 70 and all pointers initially form a correct ring. Assume that node 30 later suspects that its predecessor 10 has crashed, and 50 suspects that its successor 60 has crashed. Similarly, node 10 suspects its successor 30 has crashed, and 60 suspects that its predecessor 50 has crashed. Therefore, the system looks as if the network has partitioned into two components $\{10, 60, 70\}$ and $\{30, 50\}$. Nevertheless, node 70, might have an additional pointer to node 30, as node 70 does not suspect node 30 as crashed. If node 70 makes a lookup for the identifier 40, its request will be routed to 30, which forwards it to the responsible node 50. On the other hand, a lookup by node 10 for the same identifier 40 will be forwarded to 60, which believes that it is responsible for the identifier 40. Hence, instead of one replica of any item with identifier 4, there are two replicas, one stored at 50 and one at 60.

As demonstrated, getting a majority is problematic in a DHT. A possible way around this problem is to let nodes be conservative, and only return values when they are certain that the lookup is consistent.

Uncertainty of Lookup Consistency

The modified periodic stabilization together with atomic ring maintenance is a source of uncertainty: the initiator of a lookup does not know if the result is consistent or if it is temporarily inconsistent because of failures. We now indicate how some of this uncertainty can be eliminated by conservatively using locally available information.

If a node q's predecessor p crashes, q will detect that and set its *pred* pointer to *nil* according to periodic stabilization. In periodic stabilization, p's predecessor will at some point detect that p has crashed, and change its *succ* pointer to eventually point at q. Thereafter, q will receive a NOTIFY, which makes it change *pred* to p's predecessor. Instead of setting *pred* to *nil*, another option would be to let $q.pred$ continue pointing at p, as node q will continue to be responsible for the identifiers $(p, q]$, regardless if p has crashed or not. To facilitate failure handling, a special flag called *deadpred* could be set to true whenever the predecessor is detected as crashed.

If no failures ever occur and the failure detectors do not inaccurately report a failure, all lookups will be consistent as guaranteed by atomic ring maintenance. Any lookup for an identifier i is always forwarded until it reaches a node p for which $i \in (p.pred, p]$. Hence, the first time an inconsistency appears, it is one of the following two cases:

- Some identifiers are not the responsibility of any node. More formally, there exists some identifier i such that for every node p, it true that $i \notin (p.pred, p]$.

- Some identifiers are in the responsibility of more than one node. More formally, there exists some identifier i such that there exist two distinct nodes p and q for which $i \in (p.pred, p]$ and $i \in (q.pred, q]$.

Hence, the source of any inconsistency is due to some erroneous *pred* pointer. Therefore, if atomic ring maintenance updates a *pred* pointer, the node knows for certain that the result is correct due to a join point or a leave point. As soon as periodic stabilization changes the *pred* pointer,

the node can pessimistically assume that its lookup results might be inconsistent. More precisely, as soon as the *pred* pointer is modified in the NOTIFY procedure of Algorithm 11 (Line 28), the node can store the identifier range that is being added to its responsibility in an *Unsure* set. Similarly, if a node's responsibility shrinks with some range, that range should be removed from *Unsure*. In summary, a node p is responsible for the range $(p.pred, p]$. It is *uncertain* about the range $(p.pred, p] \cap Unsure$, and it is *certain* about the range $(p.pred, p] - Unsure$. Note that if a node detects that its predecessor has failed, it continues to be responsible for the interval between its failed predecessor and itself. It also knows that it is uncertain if it receives a lookup which overshoots the crashed predecessor.

We now motivate the use of the *Unsure* set by an example. Figure 8.1 shows a correct ring consisting of the nodes $1, 3, 5$, and 7. If node 7 inaccurately detects that node 5 has crashed, it will set *deadpred* to true. It will, however, continue to correctly respond to any lookup for the range $[6, 7]$. If it, however, receives a lookup for any identifier $[4, 5]$ from some other node, it knows that it is uncertain about those identifiers. Meanwhile, node 5 will correctly respond to lookups in the range $[4, 5]$. If node 7 eventually stops suspecting node 5 for a failure, its NOTIFY procedure will be invoked by node 5, which will make it set *deadpred* to false. Since, node 7's *pred* pointer is already pointing at node 5, its responsibility has not been extended by the invocation of NOTIFY, hence it does not add any identifiers to its set *Unsure*. Should, instead, both node 3 and 7 detect node 5 as dead, the situation will be different. In this case, node 3 will NOTIFY node 7, which will make *pred* point at 3. This implies that node 7's responsibility has been extended with the identifiers $[4, 5]$, which it will add to its *Unsure* set. Any lookup to the range $[4, 5]$ received by node 7 will result in it reporting that it is uncertain whether it is reporting a consistent result. If later node 5 is no longer suspected, it will eventually NOTIFY node 7, which will make node 7 remove $[4, 5]$ from its set *Unsure*.

Removing Uncertainty

How does a node which is uncertain about certain identifiers ever become certain. A node p which is uncertain about the range $(q_1, q_2]$ becomes certain if there exists no other correct node with identifier r in $(q_1, q_2]$. Unfortunately, determining this is difficult. For example, it might be that

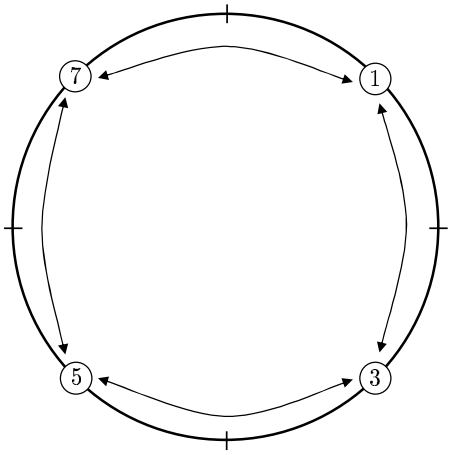

Figure 8.1: Lookup uncertainty due to a failure. Nodes 1, 3, 5, and 7 form a correct ring. If node 5 fails, node 7 continues to be responsible for the range $[6, 7]$. After 7's *pred* pointer is updated to 3, it will be responsible for the range $[4, 7]$, of which it is uncertain of the range $[4, 5]$ and certain of the range $[6, 7]$.

there exists a single node with identifier r in $(q_1, q_2]$, but due to the inaccuracy in the failure detectors, only one node m in the whole system has a long pointer directly to r. All other nodes have lost contact with r, and are no longer pointing to it. Hence, p needs to collect information from all nodes to find out that there exists some node r.

Another approach is to weaken the asynchronous model, and assume that periodic stabilization will stabilize the ring within a known time bound b. Hence, every node uses a local timer, which it resets each time the *Unsure* set grows. If the timer's value exceeds b, it knows that it can set $Unsure = \emptyset$ and hence be certain about lookups. In the previous example, the assumption implies that periodic stabilization will within b time units stabilize the ring, such that p finds out about its predecessor r. The bound b should be chosen such that it is highly unlikely that the ring does not stabilize within b time units. With this assumption, a node can always report if it is certain or uncertain. In rare cases where b is exceeded, lookup consistency might be violated.

The usefulness of the *Unsure* set is that a node can always correctly report to the application whether it is certain or uncertain about a lookup. This can be particularly useful if replication is used, as an application can

ignore the values of uncertain nodes.

Atomic Register Consistency

Next, we describe a stronger consistency model and hint how it can make use of the information regarding uncertainty, which is provided by the underlying lookup.

A stronger consistency model than regular registers is provided by *atomic registers* [80]. This consistency model is also known as *linearizability* [68]. Recall that every read or write starts with a request and ends with a response. These requests and responses occur at some distinct point in global time. An execution of this consistency model is always *linearizable*, meaning that all operations behave as if each read and write operation took place at some instant moment between the request and the response of the operation.

There exists a straightforward implementation of atomic registers in a message passing system [96]. The algorithm relies on using local timestamps for each value. A time stamp is simply a pair of values $\langle t, pid \rangle$, where t is some integer and pid is the identifier of a node. Initially, every node p starts with a time stamp $\langle 0, p \rangle$. A write $\langle k, v \rangle$ by a node p proceeds as follows. First, a read is done to a majority of the replicas of key k. Each of the replicas return the time stamp associated with their value of the identifier k. Node p picks the highest identifier $\langle t', pid \rangle$, and writes $\langle k, v \rangle$ with time stamp $\langle t' + 1, p \rangle$ to a majority of the nodes. A read by a node p to an identifier k works similarly. Node p consults a majority, and picks the value v with the highest time stamp t'. Thereafter, node p writes the value v with the time stamp t' to a majority of the nodes. This last step is necessary to ensure linearizability.

Our conjecture is that if the above algorithm only uses values of nodes which are certain, atomic register consistency is guaranteed. Since atomic ring maintenance ensures that the transfer of responsibilities is atomic, an ordinary join or leave will not need to communicate with a majority of nodes. To ensure that this algorithm works when a majority of the nodes are certain, the initiator of an operation needs to get a response from a majority of the nodes. This can either be done by using a reliable lookup (see Chapter 4) or by using a bulk operation similarly to the pseudo-reliable broadcast (see Chapter 5). Each responsible node can directly send its results back to the initiator using a reliable channel. Failures

only make it difficult to get a majority, as the result of uncertain nodes are discarded from the majority.

Bibliography

[1] 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, 2005.

[2] 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.

[3] K. Aberer, A. Datta, and M. Hauswirth. Route Maintenance Overheads in DHT Overlays. In *6th Workshop on Distributed Data and Structures (WDAS'04)*, Lausanne, Switzerland, July 2004. Carleton-Scientific.

[4] I. Abraham, A. Badola, D. Bickson, D. Malkhi, S. Maloo, and S. Ron. Practical Locality-Awareness for Large Scale Information Sharing. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, volume 3640 of *Lecture Notes in Computer Science (LNCS)*, pages 173–181, London, UK, 2005. Springer-Verlag.

[5] 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.

[6] L. O. Alima, A. Ghodsi, P. Brand, and S. Haridi. Multicast in DKS(N, k, f) Overlay Networks. In *The 7th International Conference on Principles of Distributed Systems (OPODIS'03)*, volume 3144 of *Lecture Notes in Computer Science (LNCS)*, pages 83–95. Springer-Verlag, 2004.

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

[8] L. O. Alima, S. Haridi, A. Ghodsi, S. El-Ansary, and P. Brand. Self-* Properties in Distributed K-ary Structured Overlay Networks. In *Proceedings of SELF-STAR:International Workshop on Self-* Properties in Complex Information Systems*, Bertinoro, Italy, May 2004.

[9] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Proceedings of 22nd International Symposium on Fault Tolerant Computing (FTCS'92)*, pages 76–84, Boston, MA, USA, 1992. IEEE Computer Society.

[10] M. Amnefelt and J. Svenningsson. Keso - A Scalable, Reliable and Secure Read/Write Peer-to-peer File System. Master's thesis, KTH/Royal Institute of Technology, Stockholm, Sweden, 2004.

[11] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 131–145. ACM Press, 2001.

[12] A. Andrzejak and Z. Xu. Scalable, Efficient Range Queires for Grid Information Services. In *Proceedings of the 2nd International Conference on Peer-To-Peer Computing (P2P'02)*, pages 33–40, Linkping, Sweden, September 2002. IEEE Computer Society.

[13] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced topics*. Wiley series on parallel and distributed computing. John Wiley & Sons, second edition, 2004.

[14] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A Layered Naming Architecture for the Internet. In *Proceedings of the ACM SIGCOMM 2004 Symposium on Communication, Architecture, and Protocols*, pages 343–352, Portland, OR, USA, March 2004. ACM Press.

[15] Bamboo. http://bamboo-dht.org/, 2006.

[16] 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.

[17] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, 1987.

[18] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS'03)*, pages 1–6. USENIX, 2003.

[19] E. Brewer. Towards Robust Distributed Systems, invited talk at the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00), 2000.

[20] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron. Virtual Ring Routing: Network Routing Inspired by DHTs. In *Proceedings of the ACM SIGCOMM 2006 Symposium on Communication, Architecture, and Protocols*, pages 351–362, New York, NY, USA, 2006. ACM Press.

[21] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica. ROFL: Routing on Flat Labels. In *Proceedings of the ACM SIGCOMM 2006 Symposium on Communication, Architecture, and Protocols*, pages 363–374, New York, NY, USA, 2006. ACM Press.

[22] M. Castro, M. Costa, and A. Rowstron. Should we build Gnutella on a structured overlay? *SIGCOMM Computing Communication Review*, 34(1):131–136, 2004.

[23] M. Castro, M. Costa, and A. Rowstron. Debunking Some Myths About Structured and Unstructured Overlays. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, Boston, MA, USA, May 2005. USENIX.

[24] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communica-*

tions (JSAC) (Special issue on Network Support for Multicast Communications), pages 1489–1499, 2002.

[25] J. Cates. Robust and Efficient Data Management for a Distributed Hash Table. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, May 2003.

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

[27] J. Chang and N. F. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems (TOCS)*, 2(3):251–273, 1984.

[28] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A case study in building layered DHT applications. In *Proceedings of the ACM SIGCOMM 2005 Symposium on Communication, Architecture, and Protocols*, pages 97–108, New York, NY, USA, 2005. ACM Press.

[29] E. G. Coffman, M. Elphick, and A. Shoshani. System Deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

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

[31] R. Cox, F. Dabek, M. F. Kaashoek, J. Li, and R. Morris. Practical, Distributed Network Coordinates. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, Massachusetts, November 2003. ACM Press.

[32] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS Using a Peer-to-Peer Lookup Service. In *Proceedings of the First Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, Lecture Notes in Computer Science (LNCS), pages 155–165, London, UK, 2002. Springer-Verlag.

[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] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the First USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*, San Francisco, CA, USA, March 2004. USENIX.

[35] S.E. Deering. Host extensions for IP multicasting. RFC 1054, May 1988. Obsoleted by RFC 1112.

[36] Z. Despotović. *Building Trust-Aware P2P Systems: From Trust and Reputation Management To Decentralized E-Commerce Applications*. PhD thesis, École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, 2005.

[37] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1:115–138, 1971.

[38] E. W. Dijkstra. Self Stabilization in spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, 1974.

[39] J. Douceur. The Sybil Attack. In *Proceedings of the First Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, Lecture Notes in Computer Science (LNCS), pages 251–260, London, UK, 2002. Springer-Verlag.

[40] S. El-Ansary. *Designs and Analyses in Structured Peer-To-Peer Systems*. PhD thesis, KTH/Royal Institute of Technology, Stockholm, Sweden, 2005.

[41] S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi. A Framework for Peer-To-Peer Lookup Services Based on k-ary Search. Technical Report TR-2002-06, SICS, May 2002.

[42] S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi. Efficient Broadcast in Structured P2P Netwoks. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 304–314, Berkeley, CA, USA, 2003. Springer-Verlag.

[43] C. M. Ellison. The nature of a useable PKI. *Computer Networks*, 31(9):823–830, 1999.

[44] H. Eriksson. MBONE: the multicast backbone. *Communications of the ACM*, 37(8):54–60, 1994.

[45] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[46] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[47] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *Proceedings of IFIP International Conference on Network and Parallel Computing (NPC)*, volume 3779 of *Lecture Notes in Computer Science (LNCS)*, pages 2–13, Heidelberg, Germany, November-December 2005. Springer-Verlag.

[48] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-Transitive Connectivity and DHTs. In *Proceedings of the 2nd Workshop on Real, Large, Distributed Systems (WORLDS'05)*, San Francisco, CA, USA, December 2005. USENIX.

[49] A. Ghodsi, L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. Self-Correcting Broadcast in Distributed Hash Tables. In *Proceedings of the 15th International Conference, Parallel and Distributed Computing and Systems*, Marina del Rey, CA, USA, November 2003.

[50] A. Ghodsi, L. O. Alima, and S. Haridi. Low-Bandwidth Topology Maintenance for Robustness in Structured Overlay Networks. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'04)*. IEEE Computer Society, 2004.

[51] 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)*. Springer-Verlag, 2005.

[52] S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM Special*

Interest Group on Algorithms and Computation Theory News, 33(2):51–59, 2002.

[53] S. Girdzijauskas, A. Datta, and K. Aberer. Oscar: Small-world overlay for realistic key distributions. In *Proceedings of the 4th International VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'06)*. Springer-Verlag, 2006.

[54] Gnutella. http://www.gnutella.com, 2006.

[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] R. Guerraoui and L. Rondrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, Heidelberg, Germany, 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] 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.

[60] GWebCache.
http://rfc-gnutella.sourceforge.net/src/gwc-1_9_4.html, 2006.

[61] R. van Renesse H. Johansen, A. Allavena. Fireflies: Scalable Support for Intrusion-Tolerant Overlay Networks. In Willy Zwaenepoel, editor, *Proceedings of Eurosys 2006*. ACM European Chapter, April 2006.

[62] V. Hadzilacos and S. Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical Report TR94-1425, Cornell University, 1994.

[63] T. L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC'01)*, pages 300–314, London, UK, 2001. Springer-Verlag.

[64] C. Harvesf and D. Blough. The Effect of Replica Placement on Routing Robustness in Distributed Hash Tables. In *Proceedings of the 6th International Conference on Peer-to-Peer Computing (P2P'06)*, pages 57–6, Washington, DC, USA, 2006. IEEE Computer Society.

[65] 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.

[66] S. Hazel and B. Wiley. Achord: A variant of the chord lookup service for use in censorship resistant peer-to-peer publishing systems. In *Proceedings of the First Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*. Springer-Verlag, 2002.

[67] J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 321–332. Morgan Kaufmann, September 2003.

[68] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[69] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the 21st Annual ACM Sym-*

posium on Principles of Distributed Computing (PODC'02), pages 213–222, New York, NY, USA, 2002. ACM Press.

[70] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware (MIDDLEWARE'04)*, volume 3231 of *Lecture Notes in Computer Science (LNCS)*, pages 79–98, New York, NY, USA, 2004. Springer-Verlag.

[71] J. Jernberg, V. Vlassov, A. Ghodsi, and S. Haridi. DOH: A Content Delivery Peer-to-Peer Network. In *Proceedings of the 12th European Conference on Parallel Computing (EUROPAR'06)*. Springer-Verlag, 2006.

[72] 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.

[73] 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.

[74] D. R. Karger and M. Ruhl. Diminished Chord: A Protocol for Heterogeneous Subgroup Formation in Peer-to-Peer Networks. In *Proceedings of the 3rd Interational Workshop on Peer-to-Peer Systems (IPTPS'04)*, volume 3279 of *Lecture Notes in Computer Science (LNCS)*, pages 288–297. Springer-Verlag, 2004.

[75] J. M. Kleinberg. The small-world phenomenon: an algorithm perspective. In *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC'00)*, pages 163–170, Portland, OR, USA, 2000. ACM Press.

[76] B. Koldehofe. *Distributed Algorithms and Educational Simulation/Visualisation in Collaborative Environments*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2005.

[77] S. Krishnamurthy, S. El-Ansary, E. Aurell, and S. Haridi. A Statistical Theory of Chord under Churn. In *Proceedings of the 4th Interational Workshop on Peer-to-Peer Systems (IPTPS'05)*, volume 3640 of *Lecture Notes in Computer Science (LNCS)*, pages 93–103, London, UK, 2005. Springer-Verlag.

[78] F. Kuhn, S. Schmid, and R. Wattenhofer. A Self-repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, volume 3640 of *Lecture Notes in Computer Science (LNCS)*, pages 13–23, London, UK, 2005. Springer-Verlag.

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

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

[81] M. Landers, H. Zhang, and K-L. Tan. Peerstore: Better performance by relaxing in peer-to-peer backup. In *Proceedings of the 4th International Conference on Peer-To-Peer Computing (P2P'04)*, pages 72–79. IEEE Computer Society, 2004.

[82] D. Lehmann and M. Rabin. On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem. In *Symposium on Principles of Programming Languages (POPL'81)*, pages 133–138, 1981.

[83] 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.

[84] M. Leslie, J. Davies, and T. Huffman. Replication Strategies for Reliable Decentralised Storage. In *Proceedings of the First International Conference on Availability, Reliabilitym and Security (ARES'06)*, pages 740–747. IEEE Computer Society, 2006.

[85] D. Lewin. Consistent Hashing and Random Trees: Algorithms for Caching in Distributed Networks. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, May 1998.

[86] 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.

[87] X. Li, J. Misra, and C. G. Plaxton. Active and Concurrent Topology Maintenance. In *Proceedings of the 18th International Conference on Distributed Computing (DISC'04)*, pages 320–334, London, UK, 2004. Springer-Verlag.

[88] 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.

[89] X. Li, J. Misra, and C. G. Plaxton. Concurrent maintenance of rings. *Distributed Computing (to appear)*, 2006.

[90] 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.

[91] 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.

[92] D. Loguinov, J. Casas, and X. Wang. Graph-Theoretic Analysis of Structured Peer-to-Peer Systems: Routing Distances and Fault Resilience. *IEEE/ACM Transactions on Networking (TON)*, 13(5):1107–1120, 2005.

[93] B. T. Loo, R. Huebsch, J. M. Hellerstein, S. Shenker, and I. Stoica. Enhancing p2p file-sharing with an internet-scale query processor.

In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, August 2004.

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

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

[96] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of 27th International Symposium on Fault Tolerant Computing (FTCS'97)*, pages 272–281, Washington, DC, USA, 1997. IEEE Computer Society.

[97] 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.

[98] 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.

[99] 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.

[100] G. S. Manku, M. Naor, and U. Wieder. Know thy neighbor's neighbor: The power of lookahead in randomized p2p networks. In *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC'04)*, pages 54–63, New York, NY, USA, 2004. ACM Press.

[101] 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.

[102] S. Milgram. The small world problem. *Psychology Today*, 2:60–67, 1967.

[103] M. Miller and J. Siran. Moore graphs and beyond: A survey of the degree/diameter problem. *Electronic Journal of Combinatorics*, (DS14):1–61, December 2005.

[104] A. E. Mislove. POST: A Decentralized Platform for Reliable Collaborative Applications. Master's thesis, Rice University, Houston, TX, USA, December 2004.

[105] A. E. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. Wallach. AP3: A cooperative, decentralized service providing anonymous communication. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004. ACM Press.

[106] A. E. Mislove, A. Post, C. Reis, P. Willmann, P. Druschel, D. S. Wallach, X. Bonnaire, P. Sens, J.-B. Busca, and L. B. Arantes. Post: A secure, resilient, cooperative messaging system. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS'03)*, pages 61–66. USENIX, 2003.

[107] P.V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035.

[108] M. Naor and U. Wieder. Novel architectures for P2P applications: the continuous-discrete approach. In *Proceedings of the 15th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'03)*, pages 50–59. ACM Press, 2003.

[109] M. Naor and U. Wieder. Know thy neighbor's neighbor: Better routing for skip-graphs and small worlds. In *Proceedings of the 3rd Interational Workshop on Peer-to-Peer Systems (IPTPS'04)*, volume 3279 of *Lecture Notes in Computer Science (LNCS)*, pages 269–277. Springer-Verlag, 2004.

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

[111] P2PSIP. http://www.p2psip.org, 2006.

[112] Host Identity Payload. http://www.ietf.org/html.charters/hip-charter.html, 2006.

[113] 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.

[114] V. Ramasubramanian and E. Sirer. The Design and Implementation of a Next Generation Name Service for the Internet. In *Proceedings of the ACM SIGCOMM 2004 Symposium on Communication, Architecture, and Protocols*, Portland, OR, USA, March 2004. ACM Press.

[115] A. Rao, K. Lakshminarayanan, S. Surana, R. M. Karp, and Ion Stoica. Load Balancing in Structured P2P Systems. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 68–79, Berkeley, CA, USA, 2003. Springer-Verlag.

[116] S. Ratnasamy. *A Scalable Content-Addressable Network*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2002.

[117] 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.

[118] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level Multicast using Content-Addressable Networks. In *Third International Workshop on Networked Group Communication (NGC'01)*, volume 2233 of *Lecture Notes in Computer Science (LNCS)*, pages 14–29. Springer-Verlag, 2001.

[119] A. Reinefeld and F. Schintke. Concepts and Technologies for a Worldwide Grid Infrastructure. In *Proceedings of the 8th European*

Conference on Parallel Computing (EUROPAR'02), pages 62–72, London, UK, 2002. Springer-Verlag.

[120] S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the Embarrassing Slowness of OpenDHT on PlanetLab. In *Proceedings of the 2nd Workshop on Real, Large, Distributed Systems (WORLDS'05)*, San Francisco, CA, USA, December 2005. USENIX.

[121] 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.

[122] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: a public DHT service and its uses. In *Proceedings of the ACM SIGCOMM 2005 Symposium on Communication, Architecture, and Protocols*, pages 73–84, New York, NY, USA, 2005. ACM Press.

[123] 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.

[124] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Chateau Lake Louise, Banff, Canada, October 2001. ACM Press.

[125] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, November 1984.

[126] D. Sandler, A. Mislove, A. Post, and P. Druschel. FeedTree: Sharing Web micronews with peer-to-peer event notification. In *Proceedings of the 4th Interational Workshop on Peer-to-Peer Systems (IPTPS'05)*, volume 3640 of *Lecture Notes in Computer Science (LNCS)*, pages 141–151, London, UK, 2005. Springer-Verlag.

[127] T. Schütt, F. Schintke, and A. Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. In *Proceedings of the 6th International Workshop on Global and Peer-To-Peer Computing on Large Scale Distributed Systems (CCGRID'06)*, page 8. IEEE Computer Society, 2006.

[128] Y. Shavitt and T. Tankel. Big-bang simulation for embedding network distances in euclidean space. *IEEE/ACM Transactions on Networking (TON)*, 12(6):993–1006, 2004.

[129] E. Sit, F. Dabek, and J. Robertson. UsenetDHT: A low overhead usenet server. In *Proceedings of the 3rd Interational Workshop on Peer-to-Peer Systems (IPTPS'04)*, volume 3279 of *Lecture Notes in Computer Science (LNCS)*, pages 206–216. Springer-Verlag, 2004.

[130] E. Sit and R. Morris. Security Considerations for Peer-to-Peer Distributed Hash Tables. In *Proceedings of the First Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, Lecture Notes in Computer Science (LNCS), pages 261–269, London, UK, 2002. Springer-Verlag.

[131] H.-E. Skogh, J. Haeggstrom, A. Ghodsi, and R. Ayani. Fast Freenet: Improving Freenet Performance by Preferential Partition Routing and File Mesh Propagation. In *Proceedings of the 6th International Workshop on Global and Peer-To-Peer Computing on Large Scale Distributed Systems (CCGRID'06)*, page 9. IEEE Computer Society, 2006.

[132] B. Stefansson, A. Thodis, A. Ghodsi, and S. Haridi. MyriadStore. Technical Report TR-2006-09, Swedish Institute of Computer Science (SICS), May 2006.

[133] I. Stoica, D. Adkins, S. Ratnasamy, S. Shenker, S. Surana, and S. Zhuang. Internet Indirection Infrastructure. In *Proceedings of the First Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, Lecture Notes in Computer Science (LNCS), pages 191–202, London, UK, 2002. Springer-Verlag.

[134] 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.

[135] 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 Service for Internet Applications. Technical Report TR-819, MIT, January 2002.

[136] 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.

[137] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, second edition, 2000.

[138] P. Triantafillou, N. Ntarmos, and T. Pitoura. The RangeGuard: Range Query Optimization in Peer-to-Peer Data Networks. In *3rd Hellenic Data Management Symposium (HDMS'04)*, June 2004.

[139] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC'95)*, pages 214–222, New York, NY, USA, 1995. ACM Press.

[140] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the web from DNS. In *Proceedings of the First USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*, San Francisco, CA, USA, March 2004. USENIX.

[141] H. Weatherspoon and J. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Lecture Notes in Computer Science (LNCS), pages 328–338, London, UK, 2002. Springer-Verlag.

[142] J. Xu. The Fundamental Tradeoffs Between Routing Table Size and Network Diameter in Peer-to-Peer Networks. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'03)*, San Francisco, CA, USA, March/April 2003. IEEE Computer Society.

[143] B. Y. Zhao, L. Huang, S. C. Rhea, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Global-scale Overlay for Rapid Ser-

vice Deployment. *IEEE Journal on Selected Areas in Communications (JSAC)*, 22(1):41–53, January 2004.

[144] L. Zhou and R. van Renesse. P6P: A Peer-to-Peer Approach to Internet Infrastructure. In *Proceedings of the 3rd Interational Workshop on Peer-to-Peer Systems (IPTPS'04)*, volume 3279 of *Lecture Notes in Computer Science (LNCS)*, pages 75–86. Springer-Verlag, 2004.

Swedish Institute of Computer Science
SICS Dissertation Series

1. Bogumil Hausman, *Pruning and Speculative Work in OR-Parallel PROLOG*, 1990.
2. Mats Carlsson, *Design and Implementation of an OR-Parallel Prolog Engine*, 1990.
3. Nabiel A. Elshiewy, *Robust Coordinated Reactive Computing in SANDRA*, 1990.
4. Dan Sahlin, *An Automatic Partial Evaluator for Full Prolog*, 1991.
5. Hans A. Hansson, *Time and Probability in Formal Design of Distributed Systems*, 1991.
6. Peter Sjödin, *From LOTOS Specifications to Distributed Implementations*, 1991.
7. Roland Karlsson, *A High Performance OR-parallel Prolog System*, 1992.
8. Erik Hagersten, *Toward Scalable Cache Only Memory Architectures*, 1992.
9. Lars-Henrik Eriksson, *Finitary Partial Inductive Definitions and General Logic*, 1993.
10. Mats Björkman, *Architectures for High Performance Communication*, 1993.
11. Stephen Pink, *Measurement, Implementation, and Optimization of Internet Protocols*, 1993.
12. Martin Aronsson, *GCLA. The Design, Use, and Implementation of a Program Development System*, 1993.
13. Christer Samuelsson, *Fast Natural-Language Parsing Using Explanation-Based Learning*, 1994.
14. Sverker Jansson, *AKL — A Multiparadigm Programming Language*, 1994.

15. Fredrik Orava, *On the Formal Analysis of Telecommunication Protocols*, 1994.

16. Torbjörn Keisu, *Tree Constraints*, 1994.

17. Olof Hagsand, *Computer and Communication Support for Interactive Distributed Applications*, 1995.

18. Björn Carlsson, *Compiling and Executing Finite Domain Constraints*, 1995.

19. Per Kreuger, *Computational Issues in Calculi of Partial Inductive Definitions*, 1995.

20. Annika Waern, *Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction*, 1996.

21. Björn Gambäck, *Processing Swedish Sentences: A Unification-Based Grammar and Some Applications*, 1997.

22. Klas Orsvärn, *Knowledge Modelling with Libraries of Task Decomposition Methods*, 1996.

23. Kristina Höök, *A Glass Box Approach to Adaptive Hypermedia*, 1996.

24. Bengt Ahlgren, *Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption*, 1997.

25. Johan Montelius, *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*, 1997.

26. Jussi Karlgren, *Stylistic experiments in information retrieval*, 2000.

27. Ashley Saulsbury, *Attacking Latency Bottlenecks in Distributed Shared Memory Systems*, 1999.

28. Kristian Simsarian, *Toward Human Robot Collaboration*, 2000.

29. Lars-Åke Fredlund, *A Framework for Reasoning about Erlang Code*, 2001.

30. Thiemo Voigt, *Architectures for Service Differentiation in Overloaded Internet Servers*, 2002.

31. Fredrik Espinoza, *Individual Service Provisioning*, 2003.

32. Lars Rasmusson, *Network capacity sharing with QoS as a financial derivative pricing problem: algorithms and network design*, 2002.

33. Martin Svensson, *Defining, Designing and Evaluating Social Navigation*, 2003.

34. Joe Armstrong, *Making reliable distributed systems in the presence of software errors*, 2003.

35. Emmanuel Frécon, *DIVE on the Internet*, 2004.

36. Rickard Cöster, *Algorithms and Representations for Personalised Information Access*, 2005.

37. Per Brand, *The Design Philosophy of Distributed Programming Systems: the Mozart Experience*, 2005.

38. Sameh El-Ansary, *Designs and Analyses in Structured Peer-to-Peer Systems*, 2005.

39. Erik Klintskog, *Generic Distribution Support for Programming Systems*, 2005.

40. Markus Bylund, *A Design Rationale for Pervasive Computing User Experience, Contextual Change, and Technical Requirements*, 2005.

41. Åsa Rudström, *Co-Construction of Hybrid Spaces*, 2005.

42. Babak Sadighi Firozabadi, *Decentralised Privilege Management for Access Control*, 2005.

43. Marie Sjölinder, *Age-related Cognitive Decline and Navigation in Electronic Environments*, 2006.

44. Magnus Sahlgren, *The Word-Space Model: Using Distributional Analysis to Represent Syntagmatic and Paradigmatic Relations Between Words in High-dimensional Vector Spaces*, 2006.

45. Ali Ghodsi, *Distributed k-ary System: Algorithms for Distributed Hash Tables*, 2006.

Index