Consistent lookup during Churn in Distributed Hash Tables

Stein Eldar Johnsen

26th August 2005

Summary

This thesis was written by Stein Eldar Johnsen beginning 15th August 2003 and delivered by 1st September 2005 with Svein Erik Bratsberg as mentor. The main topics are consistency and distributed hash tables.

One unsolved problem with distributed hash tables is consistent lookup. Various DHTs can show ≪acceptable≫ consistency ratings, but no DHT can show *no* lookup inconsistency during churn. We chose to use a *structural prevention* strategy to remove inconsistent lookup on the basis that *inconsistent lookups are a result of inconsistency in routing tables.* We define consistent lookup as a lookup that returns a correct membership state from some time *during* lookup.

Churn and especially ≪unplanned≫ membership changes may cause series of inconsistency problems if not handled carefully. The combination of a planned membership change (e.g. *join*) and an unplanned membership change (e.g. node crash causing a node to *leave*) can cause problems needing careful repairing in the systems routing tables. Table changes are necessary done in an order that guarantees a consistent view over index ownership, and makes the possibility of consistent termination at any point during execution.

Other novel solutions include fail-fast disconnected-detection, locking membership protocols and pre-join knowledge propagation. All these solutions are shown to improve consistency through analysis, and are easily adapted for ring geometry DHTs.

Accord was design to test many of the proposals made in the analysis. We built a distributed hash table infrastructure (with no hash table functionality), that used membership protocols based on the analysis results. The two main membership protocols were based on a 2-level 2-phase commit protocol for ≪join≫, and simple 2-phase commit with distributed operations from a single coordinator for the ≪leave≫ protocol.

The solutions proposed in this thesis are fit for all ring geometry DHTs, and some may be adapted for tree geometry DHTs, and some for all DHTs. All of Chord, Bamboo andPastry are good DHTs that can be used for testing the proposals, where all or most solutions are shown to be possible. Future work includes more testing, simulations and analysis of adaptations for different geometries.

Keywords; consistency, distributed hash table, membership, membership management, membership protocol, failure analysis, membership failure, lookup, consistent lookup.

Contents

List of Tables

List of Figures

1 Introduction

1. A person who has equal standing with another or others, as in rank, class, or age: children who are easily influenced by their peers.
3. Archaic. A companion; a fellow: ≪To stray away into these forests drear, /Alone, without a peer≫ (John Keats).
Dictionary.com/Peer

With time distributed systems have grown to proportions where the systems stall under their own weight. After some scale most distributed systems come to a point where adding nodes does not benefit for scale-up not speedup, and in the worst case slows down the system services significantly. But a new approach with Peer-to-Peer technology may help to solve the problem.

Peer-to-Peer (P2P) technology has many good properties for a distributed system. So far mostly used in loosely connected systems like file sharing (Gnutella[1]) and instant messaging (Jabber[2]). And a new overlay in P2P is Distributed Hash Tables (DHT), which in the beginning were mainly projects to create structured P2P search structures, but is now used as basis for many different projects.

A Distributed Hash Table is a structure meant to locate the ≪owner≫ of data items in a distributed system in a way similar to finding the container block in a ≪normal≫ hash table. DHTs have proved to give good qualities in terms of efficiency, many using $O(\log N)$ hops in locating a data owners by using similar degree with routing tables.

1.1 Problem Background

One interesting attribute of Distributed Hash Tables is how simple it is to make use of, and be able to rely on it for lookup and routing. And although most DHTs report of small to insignificant problems during churn, one problem seems still to be unresolved; consistency.

Bamboo tells of 99.99% consistent lookup under light churn (Bamboo F.A.Q. [3]), and 95% under heavy churn. It also compares Bamboo to other DHT's mentioned in the thesis, and none of these gives 100% consistent lookup. But is it possible? If so it may open more possibilities of use for Distributed Hash Tables in general. The problem becomes to identify where inconsistency comes from, and finding means to counter these situations.

1.2 Thesis Overview

In Section 2 we will take a look at theory of Peer-to-Peer Networks and Distributed Hash Tables. Then in Section 3 we outlay a detailed problem description, and set out the way we intend to solve the problem. The problem is described after the theory because of its close dependency on it. Analysis of the various problems are done in Section 4, and Section 5 describes the key parts of the system design. Section 6 discusses how the solutions worked with solving the problem, and how existing DHTs can make use of it, and what we got out of the analysis and the project.

We also append an appendix containing the description of, and an overview over the source code of Accord in Appendix A. The source code of Accord, it's JavaDoc, and thesis PDF file are also appended externally with a CD-ROM.

2 State of the Art

1. The mathematics of the properties, measurement, and relationships of points, lines, angles, surfaces, and solids.
2. Configuration; arrangement.
3. A physical arrangement suggesting geometric forms or lines.
Dictionary.com/Geometry

In the wake of the Internet, filesharing came with applications like Napster[4]. But in the legal turmoil of filesharing[5], offsprings like Gnutella[1] and Kazaa[6] came to counter some of the the legal pitfalls of filesharing. Both Gnutella and Kazaa solved the problem by eliminating the central server in Napster with a set of peers located on the user's computers. Both Gnutella and Kazaa are unstructured networks that uses some modified flooding algorithm for searching in the repositories.

Inspired by these programs the Distributed Hash Table (DHT), a structured peer-to-peer overlay system, was developed. One of the main focuses of the DHT is to provide a ≪put/get≫ overlay similar to that of a ≪local≫ hash table without needing to know about all the other nodes in the distributed system, and to be able to look up by value (or key or index) instead of by address.

In a Peer-to-Peer (P2P) system, all nodes have a set of the same overlay services, and communicate via a set of protocols to locate the owner nodes of each data item. But in order to enable a large number of participating nodes, each one can only have a small subset of all the nodes in its memory, and the challenge becomes then how to locate the right node for a given data item.

The field of Distributed Hash Tables is so new, as most works are from 2001 or later, therefor this Section is called ≪State of the Art≫ not ≪Theory≫. More or less all of the projects referred to are still active and related to DHTs.

2.1 Theoretical Background

Before the wake of the Distributed Hash Table, some theories came out from database communities that made the foundation for the theories and techniques of the DHT. The first mention of the term ≪Distributed Hash Table≫ came from Devine et al[7], which came up with a novel solution for updating a distributed dynamic hash table (DDH).

Devine stated that if each forwarded data access requests that were not ≪local≫, these would arrive at the correct position even if all nodes did not have updated and complete hash tables. The design also allowed for nodes (buckets) to arrive at any time, but was still based on using more or less complete hash tables at each node, and using standard methods for bucket splitting etcetera. The ≪Distributed Hash Table≫ became dynamic with regard to joins, but still not very efficient with rapid table changes (churn).

2.1.1 Replication Access

With the rapid growth of large distributed databases, access to nearby replicates of stored data became important. Plaxton et al[8] proposed a novel scheme for ordering the hash tables in groups of ≪nearby nodes≫. This scheme is called the PRR routing, the PRR scheme and the Plaxton scheme. We use the last of these names.

Plaxton Scheme Plaxton created a table with m columns and n rows such that $m^n = N$ ($n = \log_{\mathrm{m}} N$). The theory states that if each node could locate a nearby node (in terms of topology), for each position in the table, except it's own, it could by simple routing locate a node nearby which holds the desired *replica* with $O(\log N)$ steps.

$0xxx$	–	$2xxx$	$3xxx$	$4xxx$
$10xx$	$11xx$	–	$13xx$	$14xx$
–	$121x$	$122x$	$123x$	$124x$
1200	1201	–	1203	1204

A Plaxton Scheme Routing Table[8] for the node with index 1202. This example shows a simple 5×4 table allowing for a 625 node (from 5^4) DHT. The x stands for a random number in the index space, like $3xxx$ can be any node from 3000 to 3444. When routing from this table to the node 3433, the algorithm chooses the lowest table entry which has the same prefix as the target index. In this example it becomes the $3xxx$ node.

Table 1: Plaxton Scheme Routing Table

With a table of only 16 elements[1], a Plaxton scheme can handle a $5^4 = 625$ node distributed database. And on the same time it can have fast access to nearby nodes and hold routing tables reaching every node in the system with the same number of steps.

Some problems with the Plaxton scheme is the need for global knowledge at time of constructing the routing tables (called a *Plaxton mesh*); root node vulnerability, as each object is dependent on a root node as «storage block»; and lack of adaptability for dynamic queries for distant hot-spots[9] etcetera.

Replication with Plaxton Scheme The whole point of the Plaxton scheme is to be able to locate replicas for all buckets in a cost effective way. Nodes over the system is bound to have different network cost, and some node for each section of buckets is necessary *closer* than the others. Plaxton et al based their method on this, and on *local convergence*, which is an effect of locality based routing explained in Section 2.4.3.

If each nodes chooses the closest but correct node for each table entry in the Plaxton mesh, these nodes are quite likely to be the same node for some group of *close by* nodes. If then often accessed objects in the hash table are placed at nodes which are often used as part access points for these objects (or nodes that are often used for accessing an object replicate that object), the access time (due to node-to-node latency and hop-count) will average toward $1 \times l_{min}$, where l_{min} is the minimum latency to a node *in «direction» of the node owning the object.*

2.1.2 Consistent Hashing

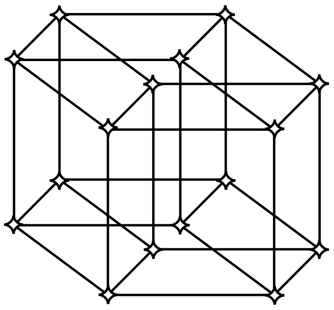
Consistent hashing[10] is a way to distribute indices over series of *ranges*. Each range is mapped into an index space ($\textbf{mod}\ I_{max}$) creating a virtual ring of ranges. To map an index to a range, each range is identified by a *bucket index*, and each index is given to its *closest* bucket for some definition of *closeness* (or *distance*).

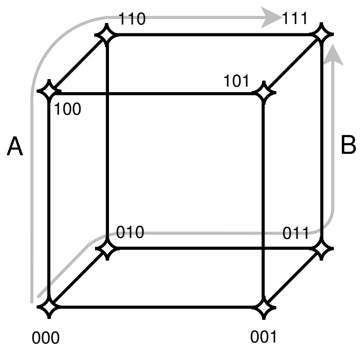
As each bucket is defined by the range from the previous bucket index to the current bucket index, each bucket can be identified by its *successor index* of the indices «belonging» to the bucket, which is also the bucket identifier.

[1]16 not 20 since the table elements at the position of the local node stands empty.

If the buckets in a hash table are positioned randomly, then given a hash function with appropriate spread and monotonicity, each bucket should contain approximately the average bucket fill. With this distribution, consistent hashing provides a true dynamic hash table structure that many DHTs are based on.

2.1.3 Hypercube


(a) A 4D Hypercube



(b) Hypercube Routing

A Hypercube is a cube of d dimensions. In the case of Figure 1(a) a 4D cube. This cube has 16 vertices and 32 edges. Each vertex is connected to 4 edges, each to different vertices in a structure that the distance between any two vertices is at most 4 edges. For a 3D cube Figure 1(b) shows how routing works. The two paths (A and B have the same length, and are thus equivalent, but they involve different intermediate nodes.

Figure 1: Hypercube

With supercomputing, one problem have been how to make a large amount of processors communicate in a fast and simple fashion, and one of the more popular solutions have been to use the hypercube[11].

A hypercube is a d-dimensional cube, the most famous being the 4-D hypercube. The hypercube is formed by making the formation of the cube as a set of dimension-adding rules, and extend it to the appropriate dimension.

Mathematically a hypercube can be described as a graph of 2^d vertices where each vertex is connected to exactly d other vertices in such a fashion that the largest distance between two vertices are d edges. Routing in the hypercube is done by ≪correcting≫ dimensions, and thus approaching the target processor. Each step in this process is deterministic, but one can choose path in order to avoid congestion at certain processors.

2.2 Distributed Hash Tables

A DHT is a vague term for describing a service for storage and retrieval of data items in a distributed system by use of a deterministic hash function, notably peer-to-peer. There are many forms of systems that uses a variety of algorithms, geometries, and communication protocols.

2.2.1 Terms

There are many terms of the DHT that must be described before the technical descriptions will make any sense. First there are some aspects of the DHT that are constantly used throughout the thesis:

DHT: *abbr.* Distributed Hash Table. Note that when referring to the DHT structure, the DHT's *geometry* or just simply ≪system≫ is often used instead.

Geometry: All references to geometry refers to the structure of which the DHT distributes and routes its indices. The geometries of DHTs are described in Section 2.3.

Topology: Topology refers to the network of nodes with communication distances (like latency, throughput etc.) that the DHT works on. The Topology of a DHT can be represented by a graph of nodes and their communication lines, which is dependent on the physical network rather than the DHT geometry or structure.

Lookup: A means to find a node in the ring that owns a specific index. In this thesis the term *lookup* will be used consistently for lookup of the owner of an index, although the same procedure is also called *routing* in some of the literature. In this thesis the two terms are used as for separate and distinguishable operations.

Routing: A means of finding a path (route) to a node closer to the owner node of an index. Routing is the basis for lookup, and is described in Section 2.4.

Neighbor: Refers to an overlay link in the routing structure (e.g. used for routing).

Distance: Most DHTs use some form of distance metric that is used for choosing routing paths, or in some cases just to describe its algorithms.

Scale: The scale of a DHT is the total number of nodes in the system. Mostly the scale is estimated or used with simulations and calculations, as it is difficult for a single node to find or calculate its true value.

Degree: The degree is the number of neighbors a node must maintain continuous knowledge of or contact with. The average or estimated degree is mainly a function of the system's scale.

Hop-count: The number of nodes that are involved in routing a message from one node to another. Most used are the *average* and *expected* hop-count of the system.

Proximity: The *proximity* of two nodes is a measurement of *closeness*, often measured with the (inverse) message round-trip latency between two nodes. Good proximity means low latency and short distance. The proximity measurement, between two nodes, is *not* related to the *distance* between the same two nodes.

There may be a little confusion about what is the *geometry* and what is the *algorithm* of a DHT. *Geometry* is used when describing the basic routing structure and the path selection algorithm of the routing strategy (e.g. Ring or Tree routing). *Algorithm* is used when describing design decisions like communication protocols, data structures or calculations (e.g. `SHA-1` or Iterative Routing).

The term *index* is also used instead of *key*. This stems from the fact that what is stored in the hash table are `"key,value"` pairs, although it is a computed *hash* value, called *index*, that is used to locate the desired bucket (or node). Therefor *index* is used to refer to the ≪numerical index value≫ used in routing, and *key* refers to the part of the data item from which the *index* is computed.

Symbols We also use some mathematical symbols to describe the algorithms.

N The scale of a DHT; the number of nodes in a system at a given point in time as a variable. Mostly used theoretically or approximately.

I_n The index of node n. Only used when a node is identified by a single index, otherwise it is explained.

I_{max} The maximum value of indices the DHT can use for its key distribution. Most calculations with indices are done ($\mathbf{mod}\ I_{max}$), representing a $\mathbb{Z}_{I_{max}}$ field of integers.

s_i The i'th successor of the *current* node.

p_i The i'th predecessor of the *current* node.

2.3 Geometries

All DHTs must organize its indices in a way that makes it possible to exactly locate stored data. That means each key must be assigned to the same *bucket* every time it is looked up by the same algorithm independent of service.

The geometry of a DHT is the way the data is organized, and how to locate the appropriate node owning each index. Since we work with numerical index values computed with hash functions, it is mostly in terms of organizing a ≪value-space≫. A range of values can be organized in a set of *dimensions*. With one dimension, we get a linear or continuous value space, more will form an d-dimensional space[12, 13]. Indices can also be arranged in a non-linear space, e.g. corresponding to a hypercube or with other models.

2.3.1 Ring

When accommodating the theory of consistent hashing into a practical DHT, The Chord[14, 15] model bay be the simplest to explain. First Chord assigns an index value to each node in the system. The value is a hash of the `"IP:port:VNID"` string (`"VNID"` is a *virtual node ID* that is used to make many virtual nodes on each physical node), using the same hash function as when calculating the indices from the data's keys.

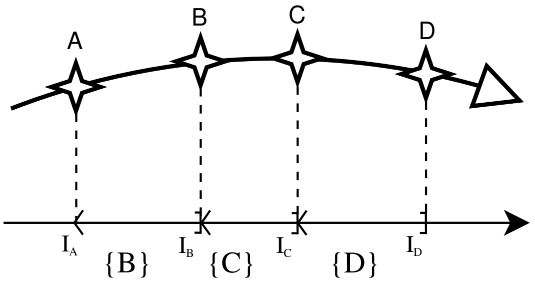
For distribution, each index is assigned to its *successor node*, that is the next node from the point of the index along the ring axis. This node is simply called the *successor node* of index i. The successor is located by routing lookup messages along the ring until the successor node of the current node is the successor of the index.

With this distribution, the distance between two nodes A and B is calculated so that the distance from A to B is $I_B - I_A$ ($\mathbf{mod}\ I_{max}$).

Other ring DHTs use a closer variant of Karger et al[10] which proposes to use the node which the largest common prefix (counted in number of bits shared before first unique) or simply numerically difference.

Skiplist routing The simplicity of consistent hashing means also a need for routing tables. Each node in Chord contains a pair of *skip-list*-like hash-tables. These tables contain the nodes at various points in the ring positioned to make maximum efficiency with a minimal table size.

The two tables used in Chord, called *fingers* and *toes*, contain the successor nodes of the indices at $I_n + 2^i \pmod{I_{max}}$ and $I_n - 2^i \pmod{I_{max}}$ for `"finger[i]"` and `"toe[i]"` respectively. The routing is done by finding the last node in the routing tables that is *not* the successor of the index sought for, and if the immediate successor is the successor node if the index, it is reported to the lookup requester as the index owner.



The ownership in a Chord ring is always allocated to the index's successor. The successor is the nodes with closest numerically *higher* index. With this example, all indices in the range from (but not including) I_B to (and including) I_C is allocated to C.

Figure 2: Ring Index Ownership

The skip-list is quite efficient, providing an average hop-count of $O(\log N)$ with a similar degree of $O(\log N)$. The efficiency is achieved by at least halving the distance to the successor node of the index for each hop.

de Brujin Kaashoek and Karger[16] showed that the hop-count gained with skip-list routing (like Chord) of $O(\log N)$ is not optimal, and that it is possible to maintain a hop-count of $O(\log N/\log\log N)$ by use of de Bruijn graphs for building its routing table when maintaining $O(\log N)$ neighbors. Koorde employs the de Bruijn graphs to maintain a constant degree DHT where the neighbor links correspond to the edges of the de Bruijn graph; two for each node in a sparsely populated ring.

Small World Symphony uses a different approach to routing. The theory is that a large mass of nodes can communicate just as efficient with a few random neighbors. For its routing table, Symphony chooses $k \geq 1$ random indices and looks up their successors. With this setup and avoiding more than $2 \times k - 1$ other such neighbor's inverse neighbors (NIN) Manku et al[17] showed that a configuration with $k = 5$ such neighbors can route very efficient independent of the scale. The system keeps an expected hop-count of $O(\frac{1}{k}\log^2 N)$, and a constant degree of k ($O(1)$).

2.3.2 Tree

The ≪old≫ way of locating a single part of a hash table is by using tree structures such as the B-tree. And with the advent of the Plaxton scheme[8], came a novel way to access a large set of nodes with a limited *local routing table*.

Zhao et al proposed such a DHT with Tapestry[9]. Tapestry uses a modified Plaxton scheme routing table, using the longest common *suffix* as distance metric. To accommodate for not being able to fill the Plaxton mesh entirely, they fill it as long as there are possible entries, and when a node cannot find a node with a longer common suffix with the index than itself, it claims the ownership.

Pastry uses also a Plaxton mesh for routing tables[18], although with using a longest common prefix as distance metric, and a ring geometry for ≪low distance≫ routing.

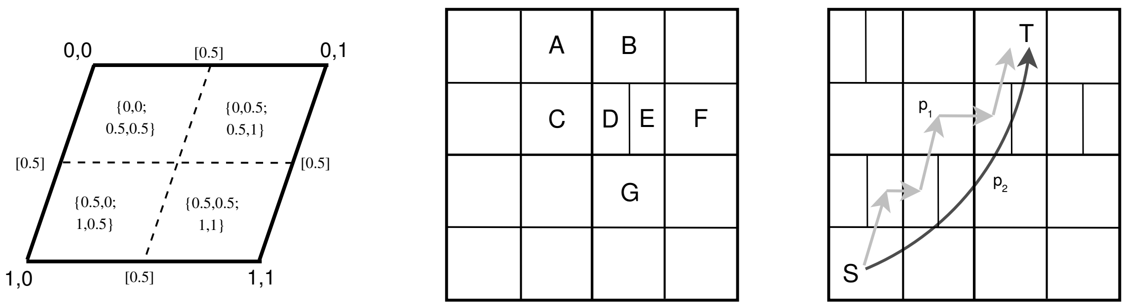
Supernodes In order to take into account the non-uniform distribution of resources among the nodes in the Internet, Brocade uses a structure of *supernodes* that form a Tapestry of their own[19]. Routing is mainly done through the supernodes thus putting less load on low-resource nodes, and more on high-resource nodes.

XOR Metric Another way to calculate distance is to use the *exclusive OR* (XOR) metric, calculated that the distance between two points x and y ($d(x, y)$) is $x \oplus y$. Note that ownership with XOR is similar to longest common prefix, although have a continuous distance metric for all index pairs. Although XOR is non-Euclidean it is very versatile in a mathematical sense[20], and by using the XOR metric Kademlia gets a unidirectional distance metric since $d(x, y) = d(y, x)$

One of the problems noted about the ring and tree model is the model's one way knowledge about it's neighbors. This results in that the nodes have to employ an active keep-alive (i-am-alive) algorithms[21]. Kademlia[20] uses a *constrained gossiping* stabilization algorithm to achieve a stable DHT structure, by fetching nodes by their lookup messages.

For routing Kademlia employs a Plaxton scheme-like routing table with entries for each $2^i \dots 2^{i+1}$ distance range, and a list of k nodes per entry (called a *k-bucket*). This make an $O(\log N)$ average hop-count and similar degree DHT.

2.3.3 Hyperspace



(a) Index Ownership (b) Neighbors (c) Routing

Subfigure 3(a) shows how the nodes in CAN divides the index space with its dimensional range coordinates. Node D in Subfigure 3(b) has 4 neighbors; B, C, E and G, likewise E has B, D, F and G as neighbors. Subfigure 3(c) shows routing from S to T. It also shows that optimal routing is not only to route in the right direction; if there is a random distribution of split nodes, they pose a difficulty in determining the distance gained for a hop.

Figure 3: Content-Addressable Network

It is possible to use a hypercube as a basis for a geometry for DHTs as well as a model for supercomputers. With CAN[13] the model has been accommodated into a more versatile indexed d-dimensional *hyperspace*. Which is done by using each dimension as a continuous index space instead of as a binary choice. Note that CAN is using a hyperspace, *not* a hypercube geometry, but if there are 2^d nodes in the DHT and uniform distribution, it is *geometrically equivalent* to a hypercube.

Distance in hyperspace is measured geometrically, so that in a 2D-space (known as Euclidean geometry), the distance between two points A and B, each measured with two axis positions, noted as A_1 and A_2 for node A and becomes $\sqrt{(A_1 - B_1)^2 + (A_2 - B_2)^2}$, and similar for more dimensions.

Nodes are not assigned space in CAN as in other DHTs. When a node enters CAN, it first locates a node that is able (or willing) to *split* its space. CAN always tries to find a node with a ≪large≫ index space to split, and thus keeps some load balancing. The old node splits its space in two halves, and gives one part to the new node.

Routing tables in CAN consists of all nodes adjacent to the local node's index space, mostly two for each dimension of the hyperspace, and routing chooses the neighbor in a node's routing table with the smallest distance to the target index, or simply a node that is closer in some dimension. This makes up for a hop-count of $(d/4) \times N^{1/d}$ (equivalent to $O(\sqrt[d]{N})$), with $2 \times d$ neighbors ($O(1)$).

2.3.4 Hybrid

It is argued which geometry is better in various aspects [see most references], but by using both consistent hashing with successor nodes, and a Plaxton scheme routing table, Pastry gains both the properties of the ring and tree geometries[18]. This is simply called the *hybrid* geometry[22].

Pastry holds a three part neighbor set. The *leaf set* is a set of successors and predecessors like the successor and predecessor lists in Chord, a Plaxton scheme *routing table* (called Plaxton mesh), and a *neighborhood set*. The leaf set ensures correct routing of messages to the closest nodes both ways in the ring, and the routing table creates a proximity aware routing table and achieving a hop-count of $\log_{2^b} N$ ($O(\log N)$), with a *fill grade* of approximately $\log_{2^b} N$ for each routing table column (see Section 2.1.1).

Bamboo is partly based on Pastry[3, 23], and also uses a hybrid ring and tree geometry DHT.

2.4 Distributed Lookup

With a lot of nodes with only partial overview over the set of nodes in the system, a DHT needs certain strategies for lookup to make it not only efficient in terms of hop-count, but also in terms of availability, proximity (latency) and consistency. The common problem is what is making the lookup fast, reliable and secure.

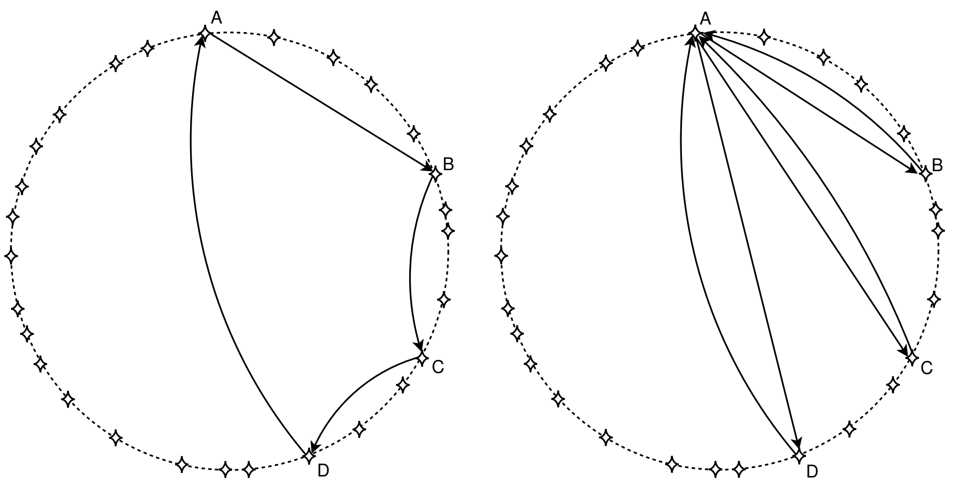
2.4.1 Iterative vs Recursive Lookup

There are different approached to how to conduct a node lookup. In case of Chord, the source node looks up the next step in its own routing tables, then ask that node of its ≪next step≫, and continues in that fashion. This is called *iterative lookup*, as the requesting node iterates to the ≪next≫ node until the successor node is found[23].

Iterative routing sends replies to each lookup request with a closer node or until the correct node is reached. This produces up $h \times 2$ message transfers, where h is the hop count.

Recursive routing is closer to the description of the routing theory itself. Each message that is not bound for the local node is forwarded to the known neighbor that is closer to the target index and is forwarded and reprocessed there. This scheme uses less message transfers than iterative routing ($h+1$), thus giving less latency, but might give greater latency penalties on failures.

Two of the systems compared in Rhea et al[23] are Chord and Bamboo. Chord is described to use *iterative* routing, versus Bamboo that uses *recursive*. The difference between iterative and recursive lookup is shown in Figure 4.



(a) Recursive Routing (b) Iterative Routing

With recursive routing the message is forwarded from node to node each time to the neighbor of the *current* node that is closest to the target index. Iterative routing first looks up the target node, and then sends the message there. Recursive routing uses $h + 1^2$ message transfers to send the message and iterative routing uses $h \times 2$ transfers.

Figure 4: Iterative and Recursive routing

2.4.2 Timeout Calculation on Lookup Messages

Rhea et al[23] argues that one of the most significant factors in lookup latency is the timeout calculation for each lookup messages. Each time a node sends a lookup message, it must wait for a reply message, and if the message don't arrive, the lookup message must either be resent, or aborted. If the timeout is too short, many successfull lookup messages will time out, and be unnecessary duplicated; and a too long timeout will result in lot of time wasted when the message is lost.

Using fixed timeouts does not take into account different loads and network congestion, and the distance between computers on the Internet.

Chord uses a virtual coordinate system for calculating timeouts[23]. The Vivaldi coordinate system uses an estimated round-trip time (v) and an average error rate(α), then applies it to the formula $RTO = v + 6 \times \alpha + 15$ (ms). The 15 millisecond constant is to avoid unnecessary retransmissions to localhost. RTO stands for round-trip timeout.

Bamboo on the other hand holds a set of average round-trip times for each neighbor, and a variance. This is the same as for TCP message timeout calculation, hence called the TCP timeout calculation. When applied to the formula $RTO = AVG + 4 \times VAR$ it is showed to produce better latency measures during very high churn rates[23].

2.4.3 Proximity Neighbor Selection

When building the routing tables some of the geometries have the possibility to choose which nodes to put there[22, 24, 25]. The property usually used there is the node's proximity. The only hurdle here is how to find the right node to put there, as no node has total overview, often even over its own neighborhood.

The ≪perfect≫ Proximity Neighbor Selection (PNS) algorithm described in [25] is based on the metrics proposed in [8]. PNS probes all the possible nodes for each table entry, and selects the most suitable node, and uses that node until a better node is located. This algorithm is mostly theoretical, as it would require the complete knowledge of nodes to choose from for each routing table entry.

There are several methods of acquiring knowledge of low-latency nodes (high proximity) without flooding the DHT with latency probes; including *constrained gossiping*[25] and *PNS(k)*[22]. Proximity Neighbor Selection has been proved a good improvement for lookup latency [26, 23, 27, 25].

Constrained Gossiping *(PNS-CG)* Constrained Gossiping is based upon spreading routing table entries on events like node join and failure. When a node joins, it collects routing tables from each of the step while looking up its own location in the DHT. When the node then has joined locally, it sends its routing table to each of the nodes in its own tables, and thus sends ≪gossip≫ about itself, and the nodes that happend to be in its tables. When a node receives a gossip message, it probes the nodes there and the sender to find possible better entries for its tables.

Constrained Gossip as described in [25] employs a *lazy* gossip protocol on node failure, where failed node entries are ignored, and the lookup requests just sent to a numerically closer node.

This model is quite constrained because it is dependent on churn (node joins, leaves and failures) to spread normal gossip. In order to avoid too little gossip on low churn, the nodes uses a timer, so that if there has been no table changes for a certain period, it will trigger a spread of local knowledge as it is.

PNS(k) [22] PNS(k) is a Proximity Neighbor Selection Algorithm that uses random sampling. The name derives from that it always selects a group of k consecutive nodes starting with the first node in the subset to select from. If one of the k nodes has a better proximity rate than the current entry, then it is substituted.

Local Convergence With a good PNS implementation there has been shown that messages from topologically close nodes that wants to route messages to the same ≪remote≫ node will *converge* and eventually be routed through a common node close to the source *group*[22, 25] of nodes.

This is called *Local Convergence* or *Locality Routing Convergence*, and can be exploited to locate replicas close to nodes that use them, and still minimize the number of replicas. This is also some of the effect used in the Plaxton scheme[8] (see Section 2.1.1).

2.4.4 Proximity Routing Selection

Another way to enhance routing performance is if a node is able to select among a number of *equally valid* routing options. Equally valid means that a choice does not hamper the hop-count or distance resulted in any significant way compared with the other choices.

The hyperspace geometry can employ a Proximity Routing Selection (PRS) algorithm[22] that finds the dimension most suitable for *fixing* by choosing the closest node (by proximity) that is closer to the target (by distance). It can do this properly because unless the target node is in the routing table of the source, there are approximately n nodes with the same hop-count (n) to the target. If for each step in the routing the closest node is chosen, then the probability of unnecessary long jumps is reduced to a minimum.

2.5 Security

Security is a serious issue for all distributed systems. In case of a node that behaves inconsistent with the protocols of the DHT or has malicious intends; what are the problems, and what can be done about it. Sit and Morris considers some issues in [28], which will be discussed here.

2.5.1 Routing Attacks

One of the (sometimes) simplest attack on DHTs is routing attacks. A node may intercept a routing request, and give a wrong response to the lookup source. This attack can do three things; firstly it can pretend that data (possibly) existing in the DHT is not there; claim ownership of the index and give false data; and send the requests to faulty or malicious nodes. But if a node wants to alter the result of routing requests in a DHT, there are measures the system can take in order to prevent it.

To prevent a node from claiming ownership to an index the correlation between index and owner should be verifiable. If a node claims to be owning an index, but the index is determined to be ≪too far≫ from the claimer, then it can be dismissed. In Chord this is quite hard to prevent as usually the nodes are so far from the lookup source that it cannot check it securely against any known threshold.

Another measure is the use of trust. A trusted node is a node that is consistently giving sound lookup results, and is keeping its share of the storage load of the DHT. This can only be measured statistically, as the ≪soundness≫ of a lookup response can be hard to calculate.

In case of *iterative* lookup, a node can monitor the progress of a lookup, and then easily detect hops that doesn't improve the routing progress or does not follow consistent behavior. And nodes that send unhelpful routing results can be marked as ≪untrusted≫.

Sadly many of these measurements can be avoided if the nodes can choose their own node IDs, as in CAN[28]. If the node ID is determined by some verifiable value like a trusted certificate, or from a hash of the node's location (like `"IP:port"`), the nodes themself cannot pretend to be anywhere else then where they are.

2.5.2 Incorrect Routing Updates

In some systems routing updates are done by gossip[20] (nodes push information about them selves and their routing tables out to other nodes). In that case a malicious node can send incorrect table updates around the system, and can direct routing to other malicious nodes or to bogus or overloaded nodes.

An effective countermeasure against incorrect routing updates is to be able to verify the table updates, either by certificates, verifiable IDs by hashing, or by probing the contents before altering the local routing tables.

2.5.3 Partitioning

A quite subtle attack is to be able to ≪snatch≫ new nodes that join the DHT into a separate distributed hash table. One can even in some cases build a parallell network which resembles the original with cross-over nodes (malicious nodes that reroutes joins to its own DHT), and replication of data from the original DHT to hide the existence of parallellism.

The most effective measure against this is to have verifiable addresses and certificates verifying the membership of nodes in the DHT. This requires that some trusted part exist that can issue certificates for joining nodes, and be able to monitor the network for malicious activity. This may easily become a bigger hazard against using it than the security it provides.

Another measure is to use random queries to verify lookup behavior. This doesn't prevent the nodes from behaving malicious, but merely detects it. The reason for using random queries is that it should not be possible to differentiate between a test lookup and a join related lookup.

2.5.4 Data Storage

Attacks can also be done against the data integrity of the DHT. This can be done in many ways; including fake writes or removes, interrupting replication procedures etc. In the case of preventing fake writes and removes, data ownership has to be known and managed, and only authorized nodes or users can modify or remove the data.

Replication protocols can be made quite inefficient if the replicas are not verifiable; meaning that the data can be cross-checked against a signature and a certificate. If all stored data is tagged with its owner, and signed, all nodes doing a read operation can check the result's signature against the certificate of the data owner.

There are other security considerations, but those will not be discussed here, and many of these are common with other distributed systems.

2.6 Churn

Churn is a way to describe the constant joining and leaving of nodes in a living distributed hash table. An example of churn rates from file sharing applications, nodes often stay from a few minutes up to an hour or more[23].

Churn brings in many problems and most obvious is that the state of the DHT is not ≪ideal≫[29]. An ideal state of a DHT can be described with that all nodes in the DHT have a consistent view over its neighbors, that is contained in its routing tables. If routing tables go ≪out of date≫, it may create problems of inconsistent or failed routing.

2.6.1 Measurements of Churn

Churn is a complex field. Nodes comes and goes, but to make a scientific study of the effects of churn, both the churn and the effects needs to be measured in a consistent way. There are

First Author	System Observed	Session Time
Saroiu	Gnutella, Napster	50% $\leq$ 60 min.
Chu	Gnutella, Napster	31% $\leq$ 10 min.
Sen	FastTrack	50% $\leq$ 1 min.
Bhagwan	Overnet	50% $\leq$ 60 min.
Gummadi	Kazaa	50% $\leq$ 2.4 min.

Different systems have showed different average session times. The shortest is that of *FastTrack* with half of the observed nodes staying no longer than 1 minute[23].

Table 2: Observed Session Times

several approaches, and some are discussed here.

Churn may be measured in session time, as each node's lifetime is often divided in sessions separated with minutes if not hours. Rhea et al argues that lifetime is not usable for measuring churn, as the ≪unavailability≫ of a node is often larger than its availability. Common availability is around 30%[23], and of a lifetime of days that can be many hours offline[29] between short sessions on-line.

Rhea at al uses a Poisson process to describe the event rate of the DHT, and where the ≪event rate λ corresponds to a median inter-event rate of $\ln\frac{2}{\lambda}$≫[23]. This can be used to calculate the median node session time of $t_{med} = N \times \ln\frac{2}{\lambda}$. With this median session time, we can construct a churn rate with realistic variation.

It is also shown that the behavior and performance of a DHT can be calculated as a result of a function of the three states of neighbor links, correct, wrong and failed, and a set of parameters describing the problem; the churn rate, stabilization rate and scale[30]. Krishnamurthy et al showed that these calculations can be quite correct, deviating from simulated results by a fraction.

Half-Life Another measure used in [29] is the half life at time t. The half life is the smaller of the time it takes for a DHT of scale N to add additional N nodes at time t, and the time it takes for $N/2$ nodes to leave the DHT at time t. The first is called the doubling time, and the latter the halving time if the DHT at time t.

The half life of the DHT is used as comparison for various events that occurs in a DHT, and what effect they have on the DHT, or if the DHT can handle the events.

Static Resilience Static Resilience is a way to explore the effects of churn without looking at the stabilization protocols[22]. There is always a time delay from nodes die until the routing tables all over the DHT are updated to accommodate for the change.

Static Resilience is then not a *measurement* of churn per se, but is used for measuring a system's performance during churn before its stabilizing algorithms have been able to repair the routing tables.

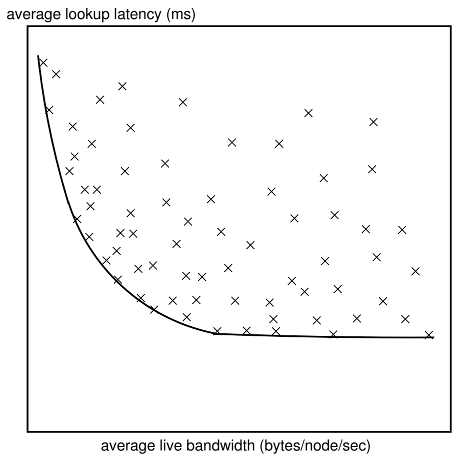
2.6.2 Effects of Churn

Some of the effects of churn is; increasing lookup latency because nodes may be forced to choose imperfect routing options[31], and in worst case being unable to make routing progress at all making them time out; and for the same reason the average hop count increases with churn; and possibly giving inconsistent lookup results[23].

These are in general ≪small≫ churn problems, with that they can be avoided by altering the algorithms and protocols surrounding the geometry. Some ≪bigger≫ problems can be described as having a *bad state*. Such states are described by having a set of nodes, each with seemingly valid routing tables, but that does still not follow the ≪rules≫ of the geometry.

One example of a *bad state* of a ring geometry DHT is the double ring[23]. Each node in the ring has a valid successor, but when following the ring at least two rounds are needed to return to the starting node. Rhea et al called this a *weakly stable* ring.

When comparing the performance of DHTs under churn, it is important to find the best cost-performance combination. In order to do this all combinations of preferences or settings of the DHT is tested, and then the *convex hull* of the testing is used as the cost-performance graph [31].



The convex hull is the line of best performances related to lookup or routing latency and bandwith usage. Bandwith is measured in average bps per node, and latency in average lookup roundtrip. Each cross represents a unique combination of settings, although this figure is *not* based on a real DHT, settings or real numbers. See Li et al[31] for correct figures.
The convex hull can be used to find optimized settings, or to measure what specific options or settings do to the efficiency of a DHT during various churn rates.

Figure 5: The Convex Hull

2.6.3 Availability

The first thought of problems with churn in DHTs is availability. Data stored in the DHT that is supposed to stay there bust be saved, and in the case of other distributed systems this is done by replication. Most DHTs use some form of replication. We will only summarize the different replication methods here.

Realities CAN is a DHT using a *hyperspace* geometry, and with availability in mind they let each node put it self in a set of *realities*[13]. The nodes places themself in different positions in different realities as to cover as much as possible *area* as possible. When some data is stored in CAN it is stored in *all* realities on the same index (coordinates), which hopefully is different nodes for each reality. Lookup from a node is done with choosing the reality where the node is *closer* to the index sought for, and use standard routing there.

Hashgroups In order to avoid the need to manage different *realities*, Pastry (or PAST) uses a collection of hash-functions called a *hashgroup*[18]. If the hash functions have good spread and distribution (load balancing and randomness), each data item should should be distributed over a set of nodes up to the number of funcitons used. This does not guarantee that the data items are replicated, but gives a very good probability they are.

Sanding Since there is problem in finding enough really good hash algorithms, and some may be not very cost effective, Tapestry manages a set of sands instead of hash functions[9]. Each tapestry chooses a set of sands prior to starting the DHT, and the data item key is sanded with each sand generating a set of indices for each item, before storing them at each index owner. With a good hash algorithm this gives similar spread as using hashgroups, but with less management, and the choice to use a single effective hash algorithm.

Successor List Chord originally used a set of successors for replicating the data items[15]. Chord uses a set of successive neighbors to a list, and use the first available successor if the original is not available. This adds a set of replicas equal to the number of successors in each node's successor list. As long as there is a live node in Chord with a replica of the sought for data item (that was in the original owner's successor list), the item is available.

Plaxton Replication The replication scheme proposed by Plaxton et al[8] is also used in Kademlia. This approach is described in Section 2.1.1 in some detail.

2.6.4 Factors and Measures

There are different effects of churn, and most factors can be countered in some way. We will summarize some of the factors and measures against them.

Update Scheduling When choosing an appropriate update scheduling for the routing tables, there are some important considerations. The bandwidth use, in the case of high churn rates, a reactive update cycle may create a lot of update messages, and this may in turn result in a *positive feedback cycle.* A positive feedback cycle is a situation where a node has a low bandwith limit, and when bombarded with update messages will appear to be offline by some nodes, and the limit is met, and message loss rate rise significantly, which will try to update all its neighbors with even more messages. Some time later (not by much, may be from a few milliseconds to minutes) the nodes are not as loaded with update messages, and reappear (as communication with other nodes are are successfull), it will again get the amount of messages forcing it offline.

Positive Feedback Cycle is a specific effect of Reactive Recovery such as used in Chord[23]. Another update cycle is to use *periodic recovery.* In this situation each nodes broadcases its table states (or changes) to its neighbors, or requests its neighbors of status changes. This method creates a stable amount of update messages, and thus avoids the positive feedback cycle.

A third way to schedule updates is to do incremental updates. Incremental updates only take one step, or a constant number of steps toward stable routing tables. This can be combined with periodic updates, which will put a limit of update messages for each cycle. On the other hand, if there has been a set of changes between two cycles, there may be table changes that are not optimal still in routing tables, until more cycles are run.

Successive Neighbors Some DHTs use *successive neighbors*, which is a set of consecutive nodes that follow the *local* node after a function of *global order.* Not all geometries support the use of successive neighbors, and the effect can be shown in terms of stability and consistency at very high churn rates[22].

Successive neighbors are shown to have a positive impact on the resilience of Distributed Hash Tables. And although not all geometries support successive neighbors by definition, some have been adapted to support them for the sake of stability.

In the case of node failures (link or otherwise)[31], shows that using a larger number of consecutive successors improves stability dramatically, although having a negative impact on table maintenance messages and bandwidth use. When studies through static resilience (Gummadi et al[22]) show that with Chord, successive neighbors generates a considerable amount of update messages, bot doesn't improve lookup latency significantly.

Gossiping Kademlia uses a ≪passive≫ stabilizing protocol, which it calls *gossiping*. The *k-buckets* holds a set of nodes, that each are collected from lookup messages. But this is not always not sufficient for keeping a set of updated node entries when some may stop responding, or the network topology changes.

The k-bucket keeps k nodes in a least recently seen (LRS) sorted list. The special feature about Kademlia is that it always keeps the *oldest* known nodes in the k-bucket when replacing entries. This stems from the propability of *continuous uptime*. If a node has been on-line for a certain time, the propability for it to still be on-line 60 minutes longer (with the same session) rises consistently with *continuous uptime*.

The update protocol used in Kademlia is to use lookups and other messages received through the ≪ordinary≫ channels to update the routing table. If there are nodes sending a message not in the appropriate k-bucket, the least recently seen node is probed for activity, and a negative response evicts it from the k-bucket, and the new node is inserted as the most recently seen node. A positive respond moves the old node to the most recently seen node for that bucket, and the new node is discarded.

2.6.5 Consistency

Liben-Novell et al talk about the problems that may lead to inconsistent behavior of a ring network [29]. There are many different problems related to the Chord network that leads to inconsistent behavior in the ring.

Liben-Nowell et al describes the *ring-like* state, where there is a core ring where the successor links construct a single ≪circle≫ around the index ring, but there may be appendages of nodes which successor links points to a node on the core ring, or to a node *closer* to the core ring in successor hops. The ring-like state is called a *soft stable* state of Chord, as if each node not on the core ring has its successor on on the core ring, it will produce consistent lookups.

Another Example is the *double ring* which is the state where if we follow the successor links we can traverse the ring at least twice and only encounter each node once. The double ring may for each node seem stable, but lookups will by a probability of about 2 to 1 return a *false* positive result.

Both of the cases from [29] have the property that they lead to inconsistent behavior in lookup and routing. Inconsistency is defined as $u.find\_successor(q) \neq v.find\_successor(q)$ here for some two nodes u and v looking for the index q.

Definitions Consistency seems to be a fairly little studied field of Distributed Hash Tables. For this reason consistency will have to be explained and discussed further in Section 3. But the topic has also been raised in the Bamboo F.A.Q.[3], where the Bamboo consistency rate is compared to other DHTs, though the definition of consistency is not discussed in [23] nor [28], although mentioning it, and showing a way to calculate it.

Rhea et al uses a simple definition of what they call a consistent lookup. They make bursts of simultaneous lookups for the exact same index (key), and treats the result as a ≪vote≫. A majority for a node is called a consistent result, otherwise it is inconsistent. This may be a too volatile definition, as it will contain both false negatives (there will naturally be a possibility of different lookup results if the sought for key just changed owner during the lookup), and false positives (if there are series of identical lookups, they may give the same wrong results, making it appear to be consistent although it is not).

Despite this Krishnamurthy[30] defines consistent lookup in terms of the node's routing tables, more specifically its immediate successor (s_1). They stated that if the immediate successor is alive and correct or not alive, it will generate consistent lookups (correct or failing), and only if s_1 is alive but incorrect will it generate inconsistent lookups. They analyzed the Chord protocol to find the states that produced these changes, and could thus calculate expected consistency rates for a given churn rate.

3 Problem Description

n. 1.a: Agreement or logical coherence among things or parts.
1.b: Correspondence among related aspects; compatibility.
2: Reliability or uniformity of successive results or events.
3: logical coherence and accordance with the facts;
Dictionary.com/Consistency

This thesis mainly looks at the possible causes of inconsistent lookup, and will try to solve those through the mean of keeping the routing tables consistent with the state of the nodes in the DHT. The assumption is that inconsistent lookup is generated by inconsistent routing tables, and that if consistency can be guaranteed over scalability, then Distributed Hash Tables will be more attractive to commercial use.

3.1 Lookup Consistency in a DHT

As stated in the Bamboo F.A.Q [3], consistency in the DHT during heavy churn is only 95%. As this is the main topic of the thesis, we must first define what we mean by the term ≪consistent lookup≫.

3.1.1 Consistency and Reliability

The term ≪consistency≫ is widely used in literature about database technology, but the four degrees of consistency described by Grey et al[32] fall short of the ≪operational consistency≫ needed for the lookup procedure. But a rather simplified consistency rule is to always access the latest value put in the database, but the term is still too vague, and doesn't take into account the reliability of the system.

Öuzu and Valduries [33] describes reliability as ≪the probability that the system under consideration does not experience any failures in a given time interval≫, and availability as ≪[...] the probability that the system is operational according to its specification at a given point in time [...]≫.

If we describe ≪lookup failure≫ as the situation where the given result is not consistent with any state of the DHT between the start and the end of the lookup procedure, then we can define specifically what we mean by ≪consistent lookup≫.

3.1.2 Defining ≪Consistent Lookup≫

We will first define ≪consistent lookup≫, and later explore the consequences of this definition. The goal is to be able to trust a lookup result to give the actual *state* of the DHT, and rather accept failed lookups than accepting the first result we can get.

It is clear that if a node does not exists in a DHT, then no node will pretend that it does. But we can work further with two failure scenarios. First is that there is inconsistent within the tables, that may return the *wrong owner.* This in turn may lead to a node believing a node that is in the DHT not to be there (breaking rule A of the rules outlined here and in Table 3).

A. The first rule states that if a node ψ is the owner of an index τ, and a data item of that key exists, then the only accepted lookup results are ψ is owner of the tuple, or that the owner is not available.

B. The second rule is rather stating that if a lookup gives a positive result, e.g. ≪the tuple sought for exists at the node ψ≫, then node ψ is the owner of the tuple (by index), and the tuple exist at that node.

(1) -	Tuple Availability States:
1 =	Owner node is ψ.
0 =	Owner node is not available.
(2) -	Predicates:
P_1	$\psi \in \Delta \cap \rho \in \Delta$
P_2	$\tau \in \Sigma$
P_3	$\psi = \texttt{owner\_of}(\tau)$
(3) -	Rules:
A	$\bigcap_{i=1..3} P_i \Rightarrow \rho.\texttt{lookup}(\tau) \in \{1, 0\}$
B	$\bigcap_{i=1..3} P_i \Leftarrow \rho.\texttt{lookup}(\tau) \in \{1\}$

The two *availability states* suggests that either the owner node of the sought for index is known, and is ρ, or it is unknown, and therefor not available.
The three predications is that first (P_1) ρ and ψ is nodes that *both are members of the same DHT*, and second (P_2) that the index τ is in the index space (Σ), and subsequently (P_3) owned by the node ψ according to the definition of ownership by the DHT.
Lastly the two rules of consistency. Rule A states that if the three predications are true; Member node ψ of DHT Δ is the owner of the index τ, and ρ is a node in the same DHT executing the lookup; then the result is either that ψ is the owner, or that the owner is unavailable. Rule B states that if the lookup results proclaims that node ψ is the owner of the index τ, then that is true for DHT Δ.

Table 3: Definition of Consistent Lookup

These rules make some implications on how consistency can be measured. Since consistency is not defined as consistent result among a group of queries, but as a reading of the ≪state≫ of the DHT, then consistency has to be checked against more information than the nodes, and to allow variations in simultaneous queries where it conscede with a state change.

The two scenarios of false positives and false negatives can be showed to break the consistency rules in Table 3. False positives will break rule B by invalidating P_1 or P_3, and false negatives will break rule B by returning the wrong state, thus invalidating P_3.

The separation of rule A and rule B is set there to allow for a negative response (e.g. unavailable owner node) as a consistent lookup restult.

3.1.3 Defining Ownership

In Section 2.6 there was a loss of a definition of what *ownership* means in terms of an evolving DHT. So when does ownership begins, and when does it ends? It is possible to define it in relation to routing (the core cycle in Chord) or completed joining.

In terms of routing a node n that ≪owns≫ some part $I_n \ \ldots \ I_{n+1} - 1$ of the index space should have the property that ownership is not falsely given to any other node as long as that ownership is in place. This will ensure that from the moment lookups can be routed to a node, and until routing will not end there later, no other lookup will end up at another node.

To put this into a time frame, when does it start? If routing is the only definition of ownership, there may be severe difficulties defining what the correct owner is at a point in time (t). Nodes come and go, and nodes may get incorrect successors (s_1).

If ownership is defined by a ≪membership≫ protocol related to a pre-join and a post-join membership state, and defines it as the node is owner *from the moment the join protocol is* *finished* and until *the leave protocol is finished.* With this definition of membership, ownership and consistent lookup, we can take a look at how to get consistent lookups.

3.2 Approaches

There are several approaches to the problem of consistency. Three ways we have considered are *algorithmic prevention*, a *statistical prevention* and *structural prevention.*

3.2.1 Algorithmic Prevention

Algorithmic prevention is to look at the lookup algorithm it self, and see if changes to it can improve consistency. One approach of this is to check a set of close neighbors to the possible owner node, and to a *neighborhood check* on ownership.

It is possible to check a series of nodes in the same neighborhood to see if they agree who is the real owner of a given index. If a majority of the nodes, including the owner, agrees, then we can say the lookup is successfull. This can be implemented as a voting protocol over a set of nodes close to τ, but this will require group access to these nodes, and this must be managed.

A problem with this protocol is the amount of lookup overhead needed. Each of the ≪voting≫ nodes must be contacted in addition with the owner node, and an agreement protocol (based on a centralized vote-commit-protocol as described by Öszu et al[33] (pp. 403–6). In addition to this, each node must use a group management protocol, to maintain a set of viable neighbors, or build on the spot a set of check-neighbors, that can verify each lookup. This adds significantly to the complexity of the lookup algorithm, and makes it not worth studying here.

3.2.2 Statistical Prevention

Simple statistics may also be used in securing the consistency of a lookup result. When doing a lookup, the DHT can do multiple lookups (l_{mul}) through different neighbors for the same index. When a minimal number of lookups ($l_{min} \geq \frac{l_{mul}}{2}$) are complete, they can be used as votes, and a majority (of $l_m ul$) will yield a presumed consistent result, and no majority will result in a presumed inconsistent result, and marked as unsuccessfully.

This approach has its weaknesses. First each lookup will have significantly longer latency as the node has to wait for enough lookups to succeed. The second is additional network traffic, linearly increased with lookup count. The third is a result of failure persistence. If a lookup is inconsistent one time, it may likely happen again, even if different neighbors are used, making it more difficult to assume consistency.

3.2.3 Structural Prevention

In database terms ≪deadlock avoidance≫ is described as to ≪employ concurrency control techniques that will never result in deadlocks or require that schedulers detect potential deadlock situations in advance and ensure that they will never occur≫ [33]. The same way we can argue that ≪lookup inconsistency avoidance≫ is to ≪employ techniques that will never result in inconsistent lookups or require lookup managers to detect potential inconsistent lookup in advance and ensure that they will never occur≫. Since inconsistency is more or less impossible to detect in advance, this will not be an option.

To detect inconsistent lookup in advance, we have to look at the routing tables, and see if there is a chance that the current tables will create possible inconsistent lookups. If we assume that *inconsistent lookup is always a result of inconsistent routing tables*, and we can eliminate *inconsistent routing tables*, then we will eliminate inconsistent lookups.

In this approach there is hopefully no extra cost for lookups, but there may be other factors and there may be other side effects. This approach is seen as the most interesting to study as it may have the potential of eliminating inconsistent lookups altogether with the right assumptions.

Thesis Approach In this thesis we chose to use *structural prevention*, as it has the potential of eliminating lookup inconsistency with no alteration to the lookup algorithms themself. It may have a cost, but it will not be explored here, as it is considerable work.

3.3 Thesis Goals

The thesis is after this section divided in three parts. First we have an analysis of various failure scenarios that can result in inconsistent lookups, and how to avoid them. Then we will try to implements those ideas into a DHT based on the Chord[15] algorithm, and then we will see what can be moved on to the existing DHTs.

3.3.1 Analysis Overview

We will analyze different failure scenarios that may result in inconsistent lookups, including single link failures, node failures and such failures in concurrency with node joins and leaves. The solutions proposed is meant to be implementable in the project named ≪Accord≫ for testing and proof of concept, and is thus based on it's geometry and algorithms. The analysis is described in Section 4.

3.3.2 Project Overview

We will also implement a DHT based on the Chord algorithm, from scratch, that is to take advantage of the solutions proposed in the analysis. The design of the system is described in Section 5. Parts of the design and implementation were done prior to analysis, and limited time has put limitation on the completeness of the implementation.

3.3.3 Discussion Overview

After the analysis and implementation, we will discuss how the solutions found and developed in Accord can be used with existing DHTs to improve their consistency rating.

4 Analysis

1. The condition or fact of not achieving the desired end or ends: the failure of an experiment.
4. A cessation of proper functioning or performance: a power failure.
5. Nonperformance of what is requested or expected; omission: failure to report a change of address.
Dictionary.com/Failure

To have a complete overview of the problem of inconsistency, we must analyze the various failure scenarios and deduce solutions that will solve the distinct failure in a consistent way. The goal is to find a set of measures to each failure scenario that will solve the problem altogether, although some of the problems may be either too complex to study thoroughly or out of the scope of the thesis.

Note that parts of the analysis was done after the initial design and implementation of the system described in Section 5. A partial redesign and reimplementation was done on basis of the analysis trying to address the problems raised here, but the system described there does *not* take into account all the failure scenarios described in this Section.

The fact that the analysis is partly based on some already taken design choices makes this section and Section 5 very intertwined. The connection between the analysis and the existing systems like Chord and Bamboo is made in Section 6.

4.1 Failure, Faults and Errors

There are different definitions of *things that can go wrong* with regard to computer software. Öszu et al[33] talks of three distinctive *non-functional* states; *failure*, *faults* and *errors*. A failure is a ≪... deviation of a system from the behavior described in the specification ...≫.

A failure waits latent in the system during *run-time*, but does not affect its *external* behavior until an *error* occurs. When a failure in the system is shown, it causes a *fault*, which is ≪... an internal state that may not obey its specification.≫ Such states shows with *erroneous behavior*, thus *failures* causes *faults* in the system state, which is shown as *errors*. Totally this is viewed as a *system failure*.

4.1.1 Consistent Failures

When lookups are executed, most are ≪consistent≫, but some are ≪inconsistent≫. To make an analysis of the causes inconsistent lookup, we must identify the *source failures* that causes inconsistent routing tables; although not all failures will generate inconsistent routing tables.

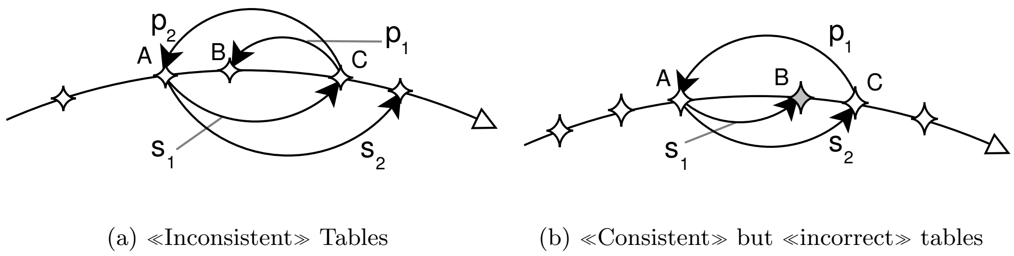
1. Lookup timeout is considered consistent, because it tells nothing of the state of the DHT, only *suggest* unavailability, and will therefor not give the impression of a non-existent (faulty) state.

2. Lookups that returns the *correct owner* of an index from any point in time during the lookup. This is from the first lookup message is sent to the result arrives at the requesting node. Any changes during this time period must be handled by the systems using the DHT, as Distributed Hast Table Storage (DHash[14], OpenHash[3] etcetera).

4.1.2 Inconsistent Failures

An *inconsistent failure* is a failure that directly or indirectly causes lookups to return wrong owner node for the index. This can be caused by a combination of some failures, or by a combination of failures and routine changes in the routing tables. The most obvious cause if inconsistency is the existence if a link to a node that is no longer in the ring by knowledge of its two immediate neighbors, and that this node acts as if still member of the ring.

Another example is that; a node joins at the same time as one or more of its immediate neighbors fails, and the subsequent stabilizing algorithm does not handle the table updates in a way that leave them consistent; or that that have states within the timespan of the membership algorithm that may cause inconsistent lookup.


p2
A
B
C
p1
s1
s2
(a) ≪Inconsistent≫ Tables
p1
A
B
C
s1
s2
(b) ≪Consistent≫ but ≪incorrect≫ tables

Showing a simple example of two table configurations that have a similar property, missing neighbor links; but behaves differently during lookup. See Section 5.1.1 for explanation of ownership and responsibility.

Figure 6: Inconsistent vs. incorrect routing tables

The two tables in Figure 6 shows situations that may give different lookup results when looking for indices between B and C given the responsibility description from Section 5.1.1. The difference is seen if the lookup traces through A or C. In 6(a) a lookup via A will believe that A is the owner of the index, but a lookup through C will forward the request to B which believes it is the owner. Both A and B are thinking they own the indices between B and C.

This situation is avoided in 6(b), as lookup through C will forward the request to A which it views as the index owner, thus the lookups will give the same result.

4.2 Communication Failures

In this thesis are communication issues mostly related to a single link that fails in some way. We will discuss network partitioning and multiple link failure, but not do a thorough analysis of its problems related to churn. When we describe ≪network link failure≫, we assume the isolated computer is still working and able to detect the failure.

4.2.1 Single Link Failure

When a single link fails, it can be transient; either for a single message transfer, or a short sequence of message transfers; or resilient, that lasts for a longer period of time.

Transient failures may be a burst of messages overloading the network, causing buffer overflows or collisions. Resilient failures may be the failure of a networking device, a broken cable, lost power at a switch etcetera; these failures cause total loss of connection, and may take minutes to hours or days to fix.

Transient Link Failure A single lost message is not considered to cause widespread inconsistency. But there is needed for some *message return timeout* when one can say that the lookup has failed. A failed lookup is not considered inconsistent (see Section 3.1.2), and we don't need to address the topic any further.

The problem arises when the link failures lasts long enough to cause the system to consider the node as unavailable or failed. It is important to note here that if a node can force itself to stop operation (self-inflicted node failure) if it detects a lasting link failure. This is called a *fail-fast* node.

To ensure that nodes rather disappears than breaks the system, nodes must be fail-fast. In the time extent a node is unavailable to the rest of the network, it must make sure it does not return believing it is a part of a ring where it is not. This may be solved by fail-fast on network activity timeout. If a node senses its own disconnectedness from the ring so long the ring may be starting to plan for its removal, the node must completely shut down, and make sure it doesn't act *connected* until it is proved to be *connected*.

`fail-fast timeout = neighbor activity timeout − security interval`

As long as the *security interval* is long enough to cover the worst case scenario with regards to when the link failed, and when the neighbor responsible of the local node's removal got the last sign of activity; the node should fail-fast before its neighbors starts considering its removal (≪forced leaving≫) from the ring.

Table 4: Fail-Fast Timeout

Resilient Link Failure Longer lasting link failures should be detected as any node or link failure for other nodes in the ring, although the node never returns within a forseeable time frame. Locally this should cause a fail-fast event, stopping the node, and the other nodes should treat it as a node failure.

4.2.2 Network Partitioning

Network partitioning, especially transient, may cause a lot of ≪inconsistency≫ in routing tables and subsequently lookups. If the partition last shorter than the *node failure timeout*, then the nodes will ≪detect≫ each other again before deciding wether to remove the nodes from the ring. Also if the partitioning lasts longer than the *node failure timeout* plus some time to consider the matter, then the ring will split in two separate rings, each trying to stabilize their own routing tables. This can cause massive inconsistency if the partitioning ends before all links between the two partitions are not removed.

With partitioning the ring may both fail and cause inconsistency. But this topic is outside the scope of the thesis, and will not be discussed further.

4.3 Node Failure

Node failure is when something *fails* locally at some node. With the assumption of inconsistency being caused by inconsistent tables, we consideres a table that causes inconsistency as a *failed routing table*. There is also the possibility that nodes simply crashes, and ceases to operate at some point.

It is probably combinations of various failures that causes inconsistent routing tables as described in Figure 6 on page 28, and such failures must be explored thoroughly. If we consider all nodes as fail-fast to all *non-trivial* failures, these failures can be viewed as a crashed node or a single link failure, and doesn't need any further discussion.

4.3.1 Node Crashes

Nodes that totally goes down is not a problem for consistency. The problem here is just to detect the failure as fast as possible, thus preventing lost lookup requests which in turn causes undesired lookup latency. This also means that a node that can detect a local failure to such an extent it may cause inconsistent lookup; it must fail totally, not partially.

4.4 Membership Failures

In the case of transient link failures, there are some protocols that are dependent on a common understanding of what the involved node *know*. Common knowledge is impossible to get without a common knowledge repository or database, which ruins the case for the DHT in the first place.

Now lets assume a set of non-described protocols that let nodes ≪join≫ or ≪leave≫ the DHT. These protocols must follow a set of predefined rules that assume some common knowledge of what the other nodes know *relative to the steps in the protocol*. These steps and knowledge assumptions depend on what failures happens, and in which place and in which order in the case of multiple failures.

4.4.1 Simple Failures

Simple failures can be that a node disappears during the run of the protocol, and to analyze this we must assume the extent of the knowledge demand before the protocol starts, during the run of the protocol, and after the protocol is done.

The Join Protocol In case of the `"join"` protocol, each state in the protocol must generate a new state of the routing tables that are considered *consistent*. As shown in Figure 6 on page 28, node links in the reverse ring must be added first in lookup ordering direction, then in the reverse lookup direction. This will result in that if the new node crashes before the link is added, then no lookups will ever end up there, and the moment the link is added, lookups will end there, and the node is a member.

What knowledge does the new node need *before* the join? Simply the knowledge needed to be a full member of the ring; a successor list that enables consistent lookups and a correct predecessor link for *keep-alive* messages. The immediate successor is needed for *any* consistent lookup, and adding the whole list increases the stability of the DHT significantly.

The Leave Protocol When nodes leave the ring, the node should be removed *last* from its predecessor, as it ensures that until the node is removed from the ring altogether, it still routes correctly to it in case of concurrent lookups. This has some problems, like nodes that only holds the node in its finger table, and does not need it for correct routing.

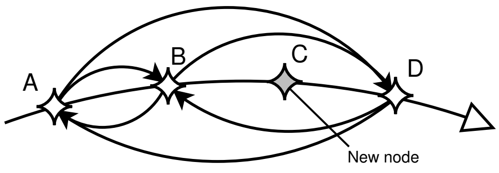
If we assume the node is fail-fast with regard to resilient link-failures, the node should have shut it self down before it is forced from the routing tables of its inverse neighbors (nodes with the ≪leaving node≫ in its routing tables). So the novel solution is to make sure the leaving node's successor removes it before it's predecessor, which has the responsibility of managing its successor.

Multiple failed nodes offers a more problematic solution, as activity knowledge for a node's s_2 is not as accurate as for its s_1. This must be taken into account when on a leave, the node's s_2 does not respond. This analysis is not done in this thesis.

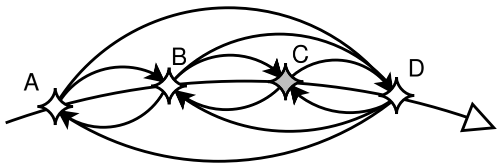
Protocol Design The two protocols we describes here and their design are found in Section 5.3, and will not be discussed further here. But there is a possibility of multiple failure, or failures related to the protocols that does not alter the protocols themselves, but some of the knowledge demands or related protocols.

4.4.2 Join-Leave Scenarios

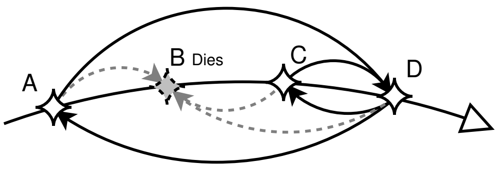
A join-leave failure is a node failure that happens during the join algorithm or closely related to the join protocol. Both scenarios described here may assume a ≪not so good≫ protocol before the analysis, but this is in order to emphasize the problem itself.



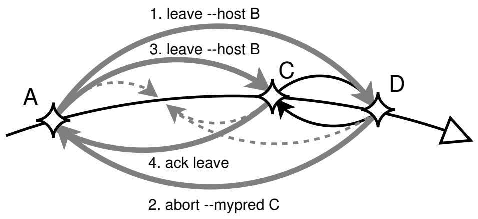
(a) Joining node C



(b) Updated tables after join



(c) B Dies and tables are rendered incorrect



(d) Leave protocol with repair collaboration

In this scenario node C joins the ring just before node B fails. This renders the table on node C problematic, but useable, and node A have a failed predecessor but doesent know that its new predecessor is C, not D. This case is the responsibility of the `"leave"` protocol on A, immediately checking for recently joined nodes.

Figure 7: Node join and failure

Failure Scenario *(A)* The first scenario is where a node join is rapidly succeeded by node failure in the new node's predecessor. This will give the new node and its new predecessor and successor some work keeping the tables both updated and consistent. The scenario is illustrated in Figure 7, and the four subfigures are used as references to states the ring is in during the scenario.

Lets trace consistency throughout the scenario. In Figure 7(a) and 7(b) the tables are both consistent and have no dead links, and therefor makes consistent lookup results. In Figure 7(c) there are introduced nodes with dead links. Lets assume node B is dead (node failure), and does not have link failure. There are now two index ranges that changes ownership during this scenario:

1. Indices belonging to C from the situation shown in Figure 7(b).
2. Indices belonging to B before the situation shown in Figure 7(c).

The other index ranges (buckets) are considered safe, as knowledge about them is consistent throughout the ring. There are also four lookup scenarios that must be checked with each of the states and possibly faulty indices.

Lookup via A *(predecessor of dying node).* A will forward both clusters of indices to B, and will subsequently fail until the leave protocol is run (see Figure 7(d)) has removed B from the successor list.

Lookup via B *(the dying node)* is correct or forwarded until B fails (see join protocol in 5.3), then all lookups will fail.

Lookup via C *(the new node)* will return correct with indices belonging to C, and indices belonging to B will fail until it is removed from the lists (fig. 7(d)). Note that this is a rare situation, as only D knows about C's membership in the ring until the situation in Figure 7(d).

Lookup via D *(successor of the new node).* Lookups for indices belonging to C will be sent to C, and are thus consistent. Indices belonging to B will fail until the protocol shown in Figure 7(d) is run, and C has updated D with new predecessor list update (see Section 5.3.4).

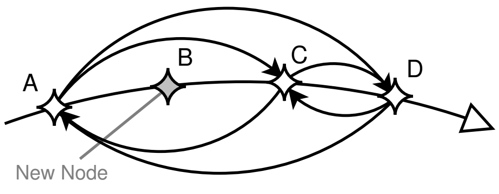
If the leave protocol is supplied with the possibility of unknown joined nodes between its known s_1 and s_2, and in the case of multiple nodes leaves at once (multiple node crashes or network partitioning), even later successors. This can be done with a recursive protocol as shown in Figure 7(d) that lets its s_2 check for its p_1 and compare. If the predecessor differs from the successor supplied with the join, it is aborted and a new s_2 is supplied.

Failure Scenario *(B)* Another scenario is if the immediate successor of a new node fails before the new node knows its s_2. This scenario is less problematic than the first scenario, as it is the responsibility of one of the nodes active with the protocol to repair the ring.

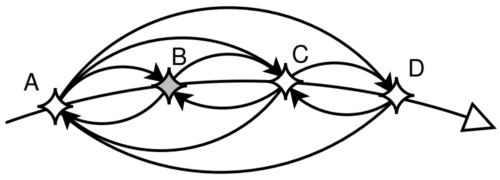
In the first scenario there was a node with almost complete tables that ≪cleaned up≫ the dead node, that made it easy because it knew its s_2 (D) when B failed. Since B doesn't necessary know its s_2 in this scenario, most lookups via this node will fail until it is repaired.

This scenario has two index clusters that are affected, but the other indices are affected as well. If we look at the effect of lookup on B's, C's and *other* indices, this should suffice for the scenario.

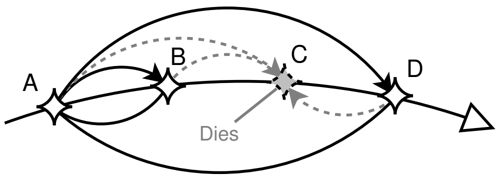
There are at least two possible solutions here.



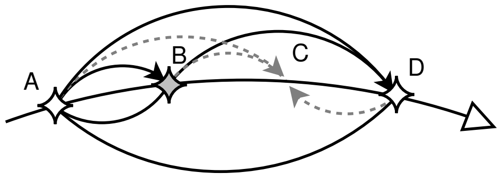
(a) Node B joins



(b) Updated tables after join



(c) C dies and tables are rendered incorrect



(d) B with additional successors

This scenario is similar to Figure 7, but the new node's successor dies instead of its predecessor. In Figure 8(d) node B is given a link to D on join, and the failure scenario can be proved to give consistent lookups with only use of commit-based (1PC) and successor-resolving leave protocol.

Figure 8: Node join and node failure combination B

1. Add a repair protocol that fixes the tables in this case. Problem: The new node must do this through its predecessor, which offer a single point of failure during the repair process. If both these nodes have failed before the routing tables are repaired, the node is still part of the ring, as some nodes *may* have added it to its routing table, but is unable to do routing nor repair its own routing tables.

2. Supply the new node with a complete successor list on time of *join*. It will now know a number of successors to query, and can take care of the *leave* protocol as if the join was not part of it.

By using additional successors, Accord is safe from problems here as long as the number of successive concurrently failing nodes does not increase beyond the number of successors the new node is supplied with.

5 Design of Accord

V. tr.: To cause to conform or agree; bring into harmony.
V. intr.: To be in agreement, unity, or harmony. Synonym [to] agree.
N.: Agreement; harmony: act in accord with university policies.
Dictionary.com/Accord

Accord was started as a small project for testing various methods and consistency related protocols on a DHT. But after a while it was clear it was becoming quite a complex Java library. Basically the Accord system was made to adhere to some usability criteria.

1. Platform Independency:
   (a) By using Java, Accord will be runnable without modification on most platforms today, including Windows, UNIX, Linux and Mac OS, though the system design itself it not dependent on the system being developed for the Java platform.
   (b) Avoiding secondary dependencies by using source based external classes only.
2. Distribution:
   (a) The DHT's theoretical distribution limit is much larger than the physical limits of todays networks for practical uses, therefor Accord should not be limited in the scale of the distributed system. Other scalability criteria are not explored in this design.
   (b) By using asynchronous communication, Accord should be very fault tolerant on link failures, and each node should be complete runtime independent of its neighbors.
3. System performance goals (No specific criteria, this depends on the protocols implemented):
   (a) Low CPU overhead.
   (b) Low Network utilization.

Accord is basically about agreement of membership. The system is made up of four services plus the lookup tables. The routing tables, are built up similar to that of Chord[15]. The four services are; the service of looking up nodes in the DHT with the ≪Lookup Service≫; managing joining and leaving of the closest successor and predecessor with the ≪Membership Manager≫; monitoring the status of the closest nodes and propagating table information along the ring with the ≪I-Am-Alive Service≫; and *stabilizing* the routing tables with the ≪Stabilizer≫.

The routing tables are described first, and the four services thereafter.

5.1 Routing in Accord

Accord is based on the *ring geometry* based on consistent hashing [10], with a *skip-list*-like routing table structure for efficient lookup modelled after Chord [15]. There are differences between Chord and Accord like indexing and the number of routing tables used.

5.1.1 Indexing

Since the objective of Accord was consistency, and not scalability, Accord is designed with simplicity rather than efficiency in mind. `SHA-1`[34] is chosen as hash algorithm because it is the basis for the Chord algorithm[15]. Both node IDs and data are indexed with `SHA-1`; nodes from the `"ip:port"` string, and data from its *key* field.

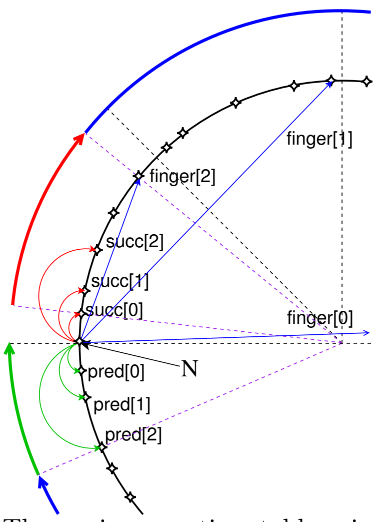
Comparing Indices Accord was built for being able to use several different means of indexing, and a special system of comparing indices was built. Since various types and ranges of indices should be comparable, and distributed over the same range of values, then both a 4 byte integer and a 40 byte `SHA-1` message digest should distribute over an equivalent range of values; *even* if the numerical value of the `SHA-1` MD is far greater than the maximum value of the integer. This distinction is made to enable Accord to use different types of indexing at the same time.

If we align the bits of the two values on the most significant bit (see table 5) and fill the missing bytes with zeroes, we get two numerics with approximately the same range, and with equivalent values and value ranges.

byte[19]	byte[18]	byte[17]	byte[16]	byte[15]	...	byte[0]
byte[3]	byte[2]	byte[1]	byte[0]	... 000 ...		

Table 5: Index Byte Alignment

Index Arithmetics Adding and Subtracting indices are done as aligned and padded with zero bytes. The result then has the length of the *largest* index. A special method creating an $I_{max} \gg n$ value, called `"ImaxRshN(n)"` is added to the *IndexFactory* to create the added offset used to locate the fingers (see 5.1.2).



The various routing tables, including the successor and predecessor lists have their index ranges of ≪responsibility≫.

Figure 9: Relative Ownership

Distance in Accord The distance metric used in Accord is the reverse that of Chord. Calculated by $d \equiv I_A - I_B (\textbf{mod } I_{max})$ for the distance from A to B, and it creates a reverse Chord-like ring.

Relative Ownership As each node has a limited view of the DHT, each node distributes its immediate neighbors on a logical ring, and assigning ranges of responsibility according to that view. The logical ring uses the same distance metric, and assigns indices to the closest node. The owner of each range of responsibility is determined by the `"owner_of(`τ`)"` method.

The range of responsibility for a node n in the ring is set to be $\forall\ \tau\ \in\ I_n \ldots (I_{s_1} - 1)\ \Leftrightarrow\ n = \texttt{owner\_of}(\tau)$.

A responsibility range as this has a bad side since there is an absolute $I_n \preceq \tau \prec I_{s_1}$ range of responsibility, and a worst case scenario a node gets a successor (s_1) with index $I_n + 1$, which gives the node a responsibility of only 1 index. The good side is that it is easier for other nodes to determine the responsible node of a given index, as the range has absolute borders defined by the closest known node.

5.1.2 Routing Tables

Accord is a distributed system, and therefore it need a consistent and efficient routing table system which with it can do fast searching and maintenance. There are three routing tables, the *successor list*, *predecessor list* and *finger table*.

The tables are referred to as `"pred"`, `"succ"` and `"finger"`. Using `"table[index]"` for calling the method to fetch the content of one of the routing tables with an reference position, similar to what is used in arrays and vectors in C/C++ and Java. When referred to with negative table references, e.g. `"finger[-1]"`, means to count from the end of the list, relative to last filled element, which means that `"succ[-2]"` means the second last element of the successor list.

Successive Neighbors Each node in the ring must know its successor, as it marks the end of its responsibility range. It should also know its successor's successor, as that is the node that will take over that border if its successor fails. By using successive neighbors Accord gains stability, but is required to manage more neighbors.

In order to have a consistently updating successor list, each node is responsible of *sending its successor list to its predecessor*. This will ensure that when a node either enters or leaves the ring, it's membership change knowledge will propagate so its predecessor (etc.) will include the new knowledge into its successor lists. And in order of convenience, the predecessor list is set to be the same size as the successor list.

Routing Tables Chord uses two secondary routing tables for achieving efficiency in lookup, the *Finger Table* and *Toe Table*. But since each node does not have a full knowledge of the number of nodes in the ring, it uses the intersection with its successor or predecessor lists as determination of the size of the finger table.

Since toes and fingers are somewhat difficult to manage, Accord will only use *finger table* and not *toe table*. The finger table is a simple skip-list, and does no attempt to achieve any lower lookup latency with PRS or PNS.

5.2 Lookup Service

The lookup service is one of the main reasons for using DHTs, and has been discussed in [23, 35] and more. Accord has two lookup procedures. One for looking up directly from the overlay tables, which is needed by the *stabilizer*, and one for looking up the owner of an index in the ring.

5.2.1 Routing Table Lookup

To open the possibility of remote looks on an overlay table, direct table lookups are available. Each table lookup is based on the query syntax `"table:index"`, which corresponds to the `"table[idx]"` call, although relative indices should be allowed here. `"table"` is the name of the routing table, and `"index"` is the table vector index.

5.2.2 Index Lookup

Index lookup, or just `"lookup"` is a simple operation, but can be easily modified with *iterative criteria.* Accord uses a no-locking lookup protocol where each external participant only checks the request and either forwards or replies to the request.

Partial Result Modes (or just ≪mode≫). When looking up an index in the local routing tables, the result will have different security properties dependent on where in the tables the node is found. A node can be *self*, *neighbor*, *safe* or *unsafe.* The *self* mode is when the responding node is the owner of the index, returning it self. *neighbor* is when the node is an *immediate* neighbor, either `"succ[0]"` or `"pred[0]"`. The *safe* mode is more complicated; if the neighbor lists are marked as ≪stable≫, then a predecessor is safe, and a successor *except the last successor* is called *safe.* All other nodes are called *unsafe.*

A routing table is marked ≪stable≫ if there are no changes for a time with the argument that nodes that have stayed for some time have a higher propability if staying even longer[20]. The *safe* mode can be argued unnecessary, but the predecessor and successor lists are managed differently than the fingers, and the *safe* nodes may be the correct owner of an index with much higher probability than the *unsafe.*

Lookup Options Lookup behavior in Accord can be modified with some simple options. We can modify timeout behavior, try count, and *iteration restriction.* Timeouts in Accord are constant but manageable. This is very easy to manage and program, but may create some unnecessary timeouts and long waits.

Implementing TCP style or Vivaldi timeout calculation requires monitoring and calculating average timeout either for the system as a whole, or for each node. Constant timeouts was chosen as it does not alter the properties of the membership protocols described in Section 5.3, and are easy to manage.

Iterative Restrictions The `"--iter"` option on lookup sets the criteria for whether a node handling a lookup request should forward it to the relative owner or reply to the originator of the request. And there are two sides of the `"--iter"` argument; if ≪stability≫ is a criteria for iteration replying to a request, or if it is a criteria for recursion forwarding of a request. See table 6 for overview of arguments.

The *all* and *none* options are equal to those discussed in Rhea et al[23], called *iterative* and *recursive* lookup respectively.

The two options *neighbor* and *safe* are to tell the nodes to; reply on immediate neighbors in addition to the local node; and to reply on immediate neighbors *and nodes considered safe*, respectively. To call a node *safe* means that it is part of a successor or predecessor list that has not been modified for some time interval. This is determined by a simple modification *timestamp* which is checked against the *current* time.

The two options *no-neighbor* and *no-safe* are opposites of *neighbor* and *safe.* They tell to *recurse* on *safer* nodes, and iterate on *unsafe* nodes. The case for the *no-neighbor* is that if a node receives a lookup request that it believes are bound for its own immediate successor or predecessor it can forward it with a high probability of it still being alive.

argument	recurse	description
`all`	`l,n,s,u`	Respond on *all* requests.
`safe`	`l,n,s[u]`	Respond on all requests except on *unsafe* links.
`neighbor`	`l,n[s,u]`	Respond on local and *neighbor*, else forward.
`default`		Use default iterative restriction argument.
`no-neighbor`	`l[n]s,u`	Respond on all except when node is a *neighbor*.
`no-safe`	`l[n,s]u`	Respond when the node is *local* or *unsafe*.
`none`	`l[n,s,u]`	Only respond when node is *self*.

The ≪recurse≫ column shows what behavior to choose on a lookup given its result *mode*. The four letters stand for *local*, *neighbor*, *safe* and *unsafe*, and the letters surrounded with braces (`[` and `]`) are the modes on which to choose to ≪recurse≫ the request to the next node instead of responding to the initiator. The nodes called *neighbor* are the immediate neighbors, s_1 and p_1.

Table 6: Iterative Restriction Arguments

5.2.3 Consistent Lookup

In Section 4 we discussed consistency with lookup, and found that the lookup protocol is good enough as it is, and if the membership protocols make sure the routing tables are consistent at *all* times, it does not need to change. We will therefor not discuss the matter further.

5.3 Membership Management

In 4.4 we showed that inconsistent tables can cause inconsistent lookup results. This section looks at how to manage the joining and leaving of nodes, and the solutions supposed to solve the inconsistency problem.

We propose two protocols, a new join protocol aimed at updating the routing tables in a fashion that produces consistent tables in every step. To guarantee this the join protocol is based upon a *2-phase-commit* protocol as showed in Ouzo et al[33]. The two *Membership* protocols called ≪join≫ and ≪leave≫ are responsible to achieve this.

All the failure cases of 4.4 are shown to be handled with carefull membership management. There are two important management protocols, ≪join≫ and ≪leave≫. The *join* protocol takes a new node safely into the ring, and the *leave* protocol takes a node safely out of the ring.

In addition, the nodes have a push based keep-alive protocol, called the ≪i-am-alive≫ service. Since the finger table is not reflective, there is no help in propagating it to the members. The *stabilizer* is not a protocol per se, but a description of the dynamic building and repairing of the routing tables that are used in Accord.

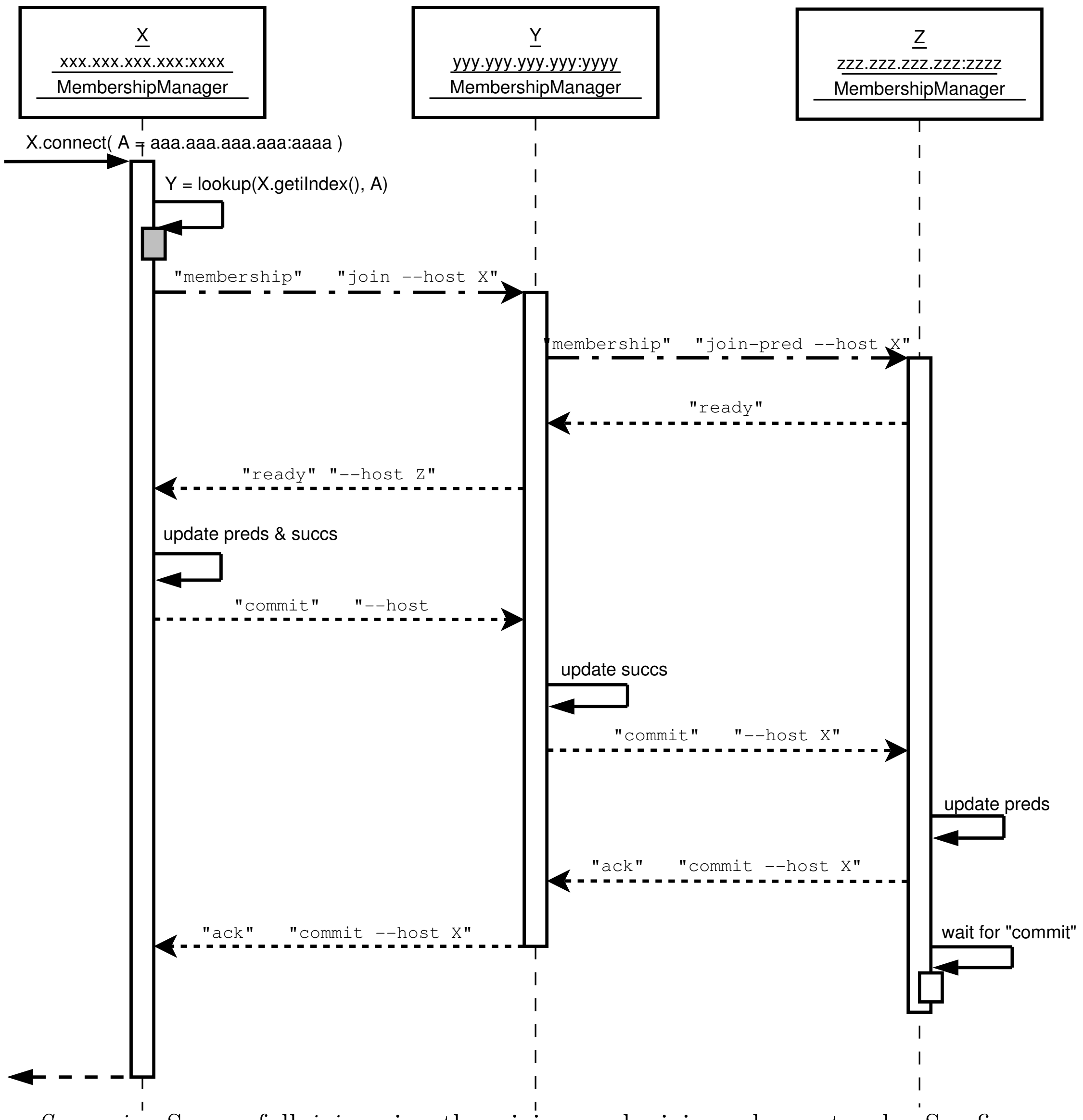
5.3.1 Neighborhood Responsibility

In Accord, each node is responsible for managing it's *successor*, and to notify it's *predecessor* of the changes in its successor list and to some degree the same for its predecessor. Since each node's responsibility is defined as the range from it's own index to it's successors index, managing the successor correctly will ensure consistent tables. This stems from that each node must keep track of the end of its *responsibility range* (5.1.1).

5.3.2 Join Protocol

The join operation should be given the ACID properties as data in a database holds. When a new member joins the DHT, it should from some moment and on be a member, for all operations done thereafter. The four properties have different solutions related to the ≪join≫ and ≪leave≫ protocols.

The ACID properties: Atomicity, Consistency, Isolation and Durability are solved with 2-level 2-phase commit join protocol and commit based leave protocol, ordered table changes, membership protocol locking and pre-join knowledge respectively. But since some of these means and effects are interdependent, they must be assigned to the protocol as a whole.



Scenario: Successfull *join* using the ≪join≫ and ≪joinpred≫ protocols. See figures 11 and 12 for flowchart showing the protocol. *Join* in the protocol between X and Y, and *joinpred* is the protocol between Y and Z.

Figure 10: Successfull Join

The join protocol is quite complex, and is described in full in a flowchart in Figure 11 and Figure 12, and as a UML sequence diagram for a successfull join operation in Figure 10. Figure 11 shows the interaction between the join protocol initiator (joiner) and the joining node's predecessor in the ring, which is the node defining its range of responsibility. Figure 12 shows the interaction with the *coordinator* of the join, which is the new node's predecessor, and its successor prior to completion. The flowchart shows the three *substates* of the coordinator; ≪JP_INIT≫, the *joinpred initial* substate; ≪JP_COM≫, the *joinpred commit* substate; and ≪JP_ABO≫, the *joinpred abort* substate.

Note that both protocols follows the two-phase commit in all ways except in termination, and in the possibility of ≪direct≫ commit notification if neighbors are already in place.

Because of the two-level structure and using consistency as the main criteria for termination, the protocol can have a much more relaxed termination protocol, based on timeouts, than standard 2PC protocols. The only importance is that the *join-pred* protocol does not terminate after sending its commit confirmation. This avoids that the confirmation message is lost and the protocol ended, thus forcing the predecessor to manually check its successors routing tables.

The ordering of table changes is shown with the ≪add≫ and ≪rem≫ enclosed operations. Locking is achieved with locking access to the join and leave protocols to only one instance at a time. This ensures that two join operations on the same node does not interfere with each other.

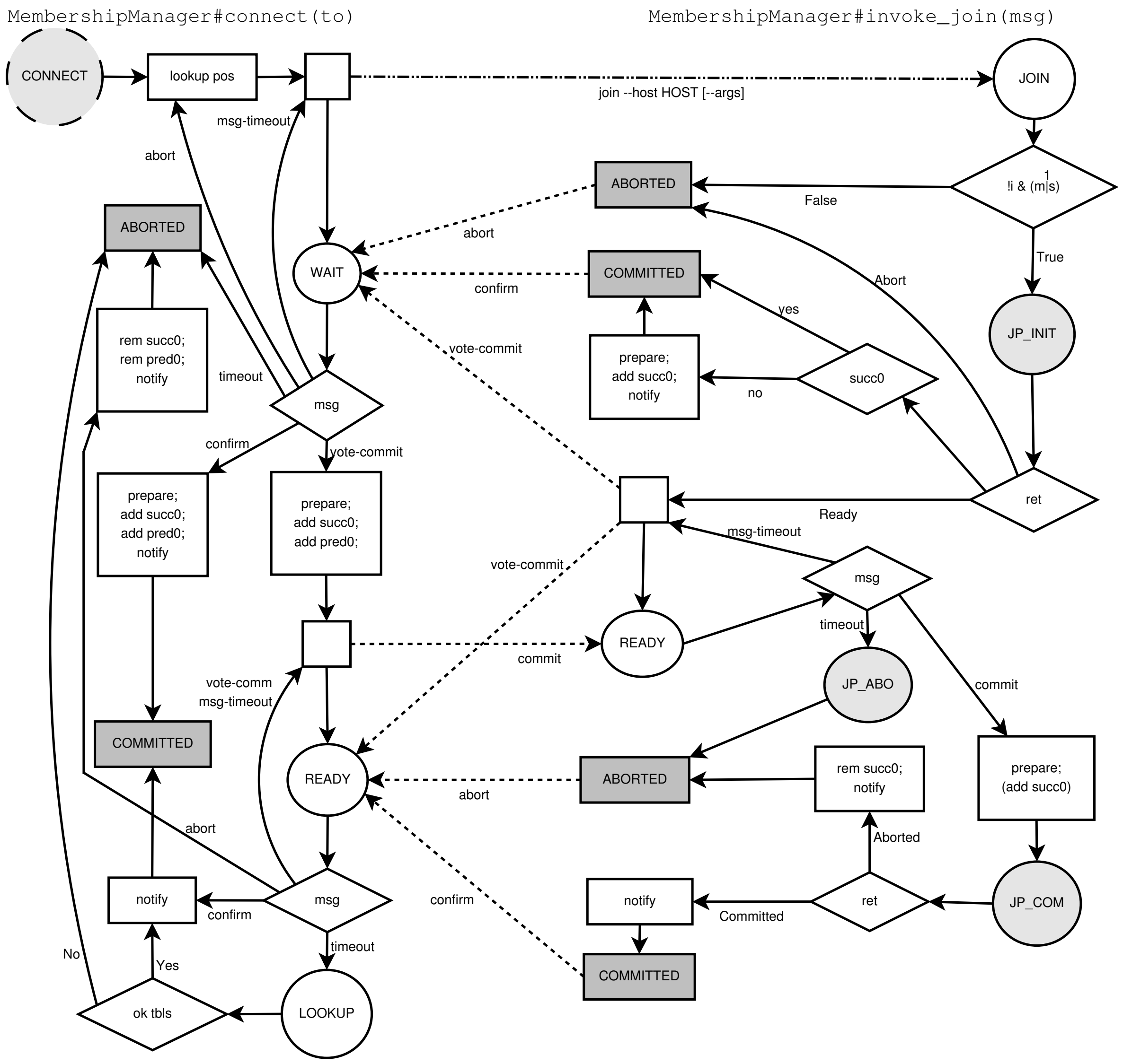
The Connector-Predecessor ≪join≫ Protocol is the part going between the initiator of the join (the joiner), and the node that is to become its predecessor. The flowchart (figure 12) follows the flow rules of state-machines, and each ≪box≫ represent either a state (circle), a decision (diamond), or enclosed operations (squares). The dotted and dashed lines are asynchronous messages sent between two processes. Light gray boxes represent other state-machines, and darker gray represent end states.

The protocol has three main phases; initialization, ready and done. When it is done it can either be committed or aborted. In the committed end-state for the initiator node it is positively added to the ring. The problem here is if the invoked predecessor fails during commit phase. The only way for the initiator to know if join has been successfull is to lookup it self in the ring. If response is positive (returns ≪self≫), the node has joined, if the response is negative (other response), the join has failed.

The Predecessor-Successor ≪joinpred≫ Protocol takes part solely between the invoked predecessor and its successor (as prior to join). On the initiators part (joiner's predecessor) there are three states, *joinpred Initiate* ≪JP_INIT≫, *joinpred Abort* ≪JP_ABO≫ and *joinpred Commit* ≪JP_COM≫ as showed in the flowchart in figure 12. In total the three states makes up the two phases of the commit protocol. The extra ≪loop≫ on the end of the successors protocol makes sure the commit is received by the initiator.

5.3.3 Leave Protocol

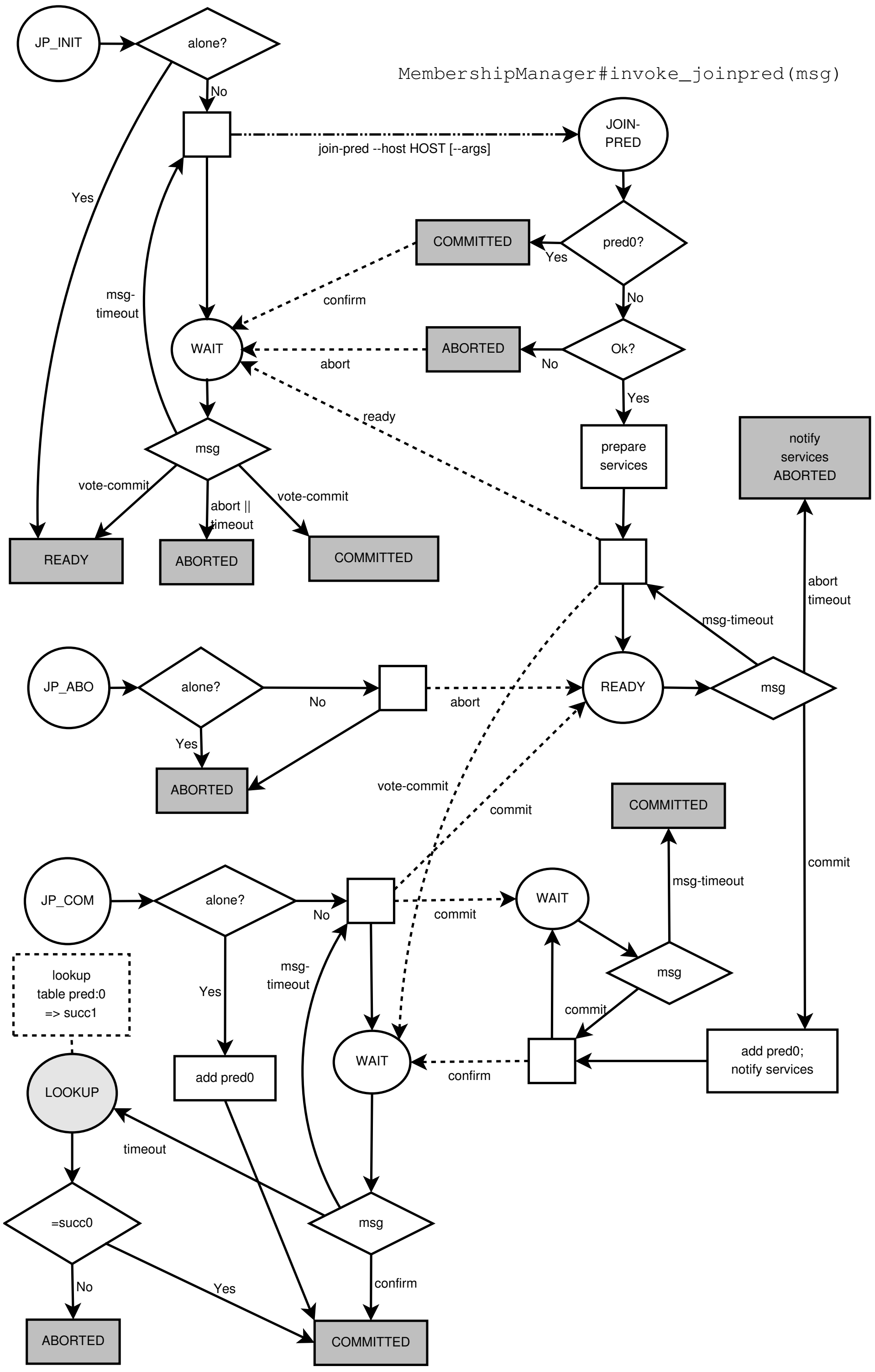
The leave protocol of Accord is similar to the join protocol, but much simpler. The second level of the protocol (between the predecessor and the successor of the leaving node) is a one-phase commit protocol, as that removal does not break consistency in lookup (see Section 5.2.3 and Section 4.4). The important part here is a node's recoverability from after being halfway



1. Composite boolean operation:
   $\neg isIndexConflict \ \wedge \ (\ indexIsMyResponsibility \ \vee \ (joiner = \texttt{succ}[0] \ ) \ ).$

Flowchart over the ≪join≫ protocol. The flow ends at the dark gray boxes (end states), and the light gray boxes are larger procedures or parts of protocols which gives a return value. The protocol follows the basics of the two-phase-commit[33].

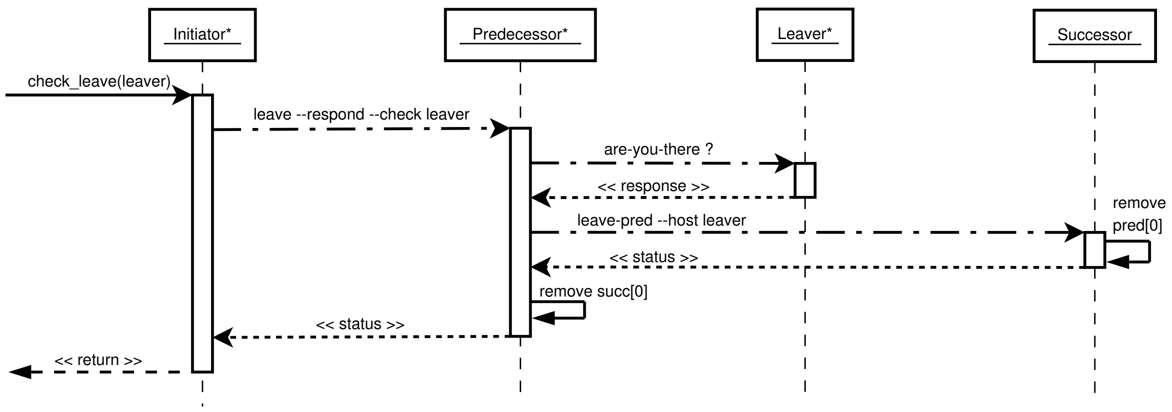
Figure 11: ≪Node Joining≫ Protocol Flowchart



The ≪joinpred≫ protocol part of the join protocol. The three states (≪JP_INIT≫, ≪JP_ABO≫ and ≪JP_COM≫) are the three states described in 5.3

Figure 12: ≪Predecessor Joining≫ Protocol Flowchart

removed from the ring. It is done by invoking the join protocol to repair links consistently. See Figure 13 for sequence of messages during the protocol. No flowchart is given.



- The Initiator can either be the leaver it self in case of ≪disconnect≫ or the leaver's predecessor in case of failed ≪i-am-alive≫ messages. See Section 5.3.4 for more on that.

The leave protocol successfull leave protocol use. This case is a mix of the failing node and disconnecting scenarios, as on ≪disconnect≫ the node is never *checked*, and on failing node it doesn't request a response.

Figure 13: Leave protocol sequence diagram

Note that the leave protocol only works if the leaver's predecessor can access the successor. In case of multiple consecutive node failures, the predecessor of the first failing node must start the leave protocol on the last failed node first, and then collapse the ring in between.

5.3.4 I-Am-Alive Protocol

The responsibility of making sure both neighbors know of the continuing activity of a node, and propagating neighbor lists throughout the ring is the ≪I-Am-Alive protocol≫.

The I-Am-Alive protocol is designed to take care of updating the predecessor and successor lists. It does that by sending messages both ways periodically with its continuing lists for the target. So the first successor get sent the predecessor list, and the first predecessor gets the successor lists.

Accord uses periodic neighbor update instead of reactive neighbor update because periodic updates are simpler to manage, and costs less network activity in case of churn. The down side is a constant network traffic cost of two I-Am-Alive messages each sending interval for each node in the ring. This is considered trivial as Accord is not designed for large scale DHTs.

The I-Am-Alive protocol will activate the ≪leave≫ protocol if it looses contact with its successor. But in the case for its predecessor (p_1), it will only *notify* its p_2 when it looses contact with its p_1. A result of this is that only the predecessor of a node can force a node to leave, and only if the node's next alive successor also have removed the node. See 5.2.2 for proof of consistency on lookup.

Counting Rules:[1]			
[1]	$tmp = ((N - \texttt{min\_succ}) \times \texttt{stable\_succ\_ratio})$		
	$FC = \texttt{if}(tmp > 0)\texttt{then}(N - \texttt{min\_succ} - tmp)\texttt{else}(0)$		
[2]	$NC = N - FC$		
Redundancy Avoidance:			
[3]	$FC = 0$	$(\texttt{succ}[NC-1] \preceq \texttt{pred}[NC-1] \prec \texttt{me})$	
	$FC > 0$	$(\texttt{me} \preceq \texttt{succ}[NC-1] \prec \texttt{finger}[FC-1])$	
Completeness:			
[4]	$FC = 0$	$NC < \texttt{min\_succ}$	$(\texttt{pred}[NC-1] \preceq \texttt{succ}[NC-1].\texttt{succ}[0] \prec \texttt{me})$
		$NC \geq \texttt{min\_succ}$	$(BI \preceq \texttt{succ}[NC-1] \prec \texttt{me})$
	$FC \geq 0$	$(FH(FC+1) \preceq \texttt{succ}[NC-1] \prec \texttt{finger}[NC-1])$	

1. These rules are calculated, not ≪checked≫ as a rule that can be broken, and the result will vary and is used in the next rules to check the actual stability criteria.

The $(\phi \preceq \tau \prec \psi)$ function is a representation of the *between*(ϕ, τ, ψ) function, notion is similar to the geometrical rule of betweenness. I_{max} is the number of values represented in an index, and BI is the index marking the ≪border≫ of when to start using fingers. All node references refer to their index values unless used for locating other nodes. The $FH(i)$ method refers to the ≪`finger[`$i-1$`]` is owner of≫ property.

Table 7: Stabilization Rules

5.3.5 Stabilization

Accord uses a periodic routing table updating protocol. The difference between periodic and reactive table updates are discussed in [23]. In Accord long periodic update of non-critical tables are used, as it requires less monitoring structure like leave notification, but it costs some continuous network traffic, and greater risk of outdated tables.

Since the *membership manager* takes care of consistent neighbor joins and leaves, and the *i-am-alive* protocol quickly updates the neighborhood lists after such updates, only the long range overlay links, and *stabilizing* the lists are left. This is the responsibility of the *stabilizer* process.

In Table 7 the method $FH(n)$ is used. It is a function that generates the index value which each fingers should be the owner of respectively.

The stabilizer have two main functions:

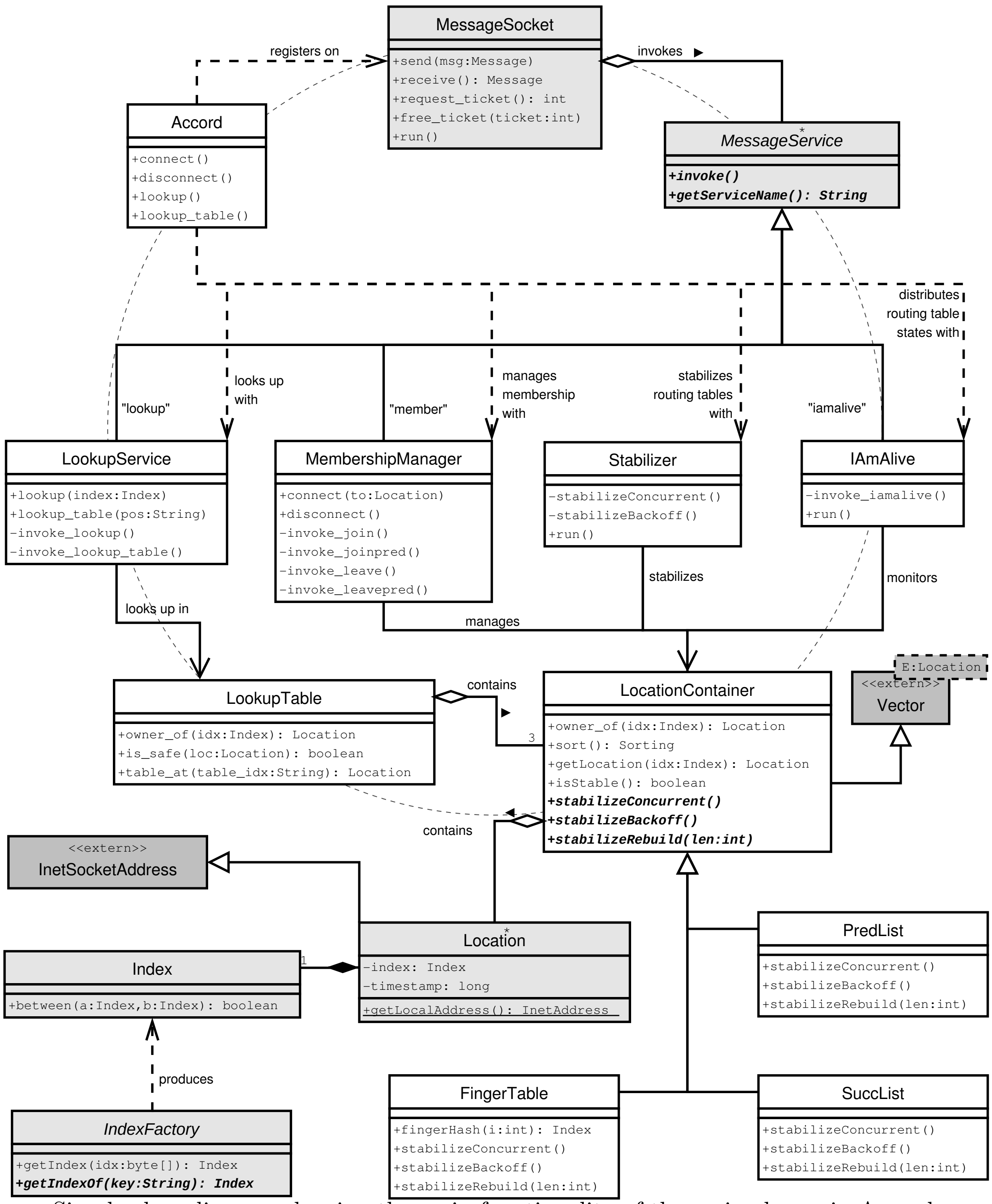
1. Make sure all links in the overlay table are valid (alive and a member of the same ring).

2. Stabilize the ratio of successors to fingers according to the current table structure.

To make a simple algorithm for achieving *(2)* we have made some simple rules (see table 7) that are run periodically to check if the tables are over-extending, insufficient or good. Over-extending tables are reduces, insufficient are increased and good tables are kept as-is.

5.4 System Design

All in all, there are four main services of Accord: the *Lookup Service*, the *Membership Manager*, the *I-Am-Alive Service* and the *Stabilizer*. These work on a set of *Overlay Containers* which makes up the overlay table or *Lookup Table.*



Simple class diagram showing the main functionality of the main classes in Accord. The four light gray classes are not part of Accord per se, but are important in terms of functionality. The two darker gray classes are standard in Java.

Figure 14: Accord Class Diagram

6 Discussions and Conclusions

A: n. (r^{oo}t, rout)
1. A road, course, or way for travel from one place to another.
2. A customary line of travel.
B: tr.v. routed, routing, routes
1. To send or forward by a specific route.
2. To schedule the order of (a sequence of procedures).
Dictionary.com/Routing

The analysis shows that it is theoretically possible to construct a ring geometry DHT that avoids many of the traps of inconsistent routing tables. And the design and implementation of Accord helped develop series of remedies, and clean up the analysis part. The problem of consistency is not solved in general for DHTs, as these remedies and solutions must be adapted for each DHT, and there may be DHTs that can not use some or any of them.

This section discusses the future of this work, and how it fits with the existing DHTs like Chord, Tapestry, Kademlia etcetera. We will also discuss what it really solves and how those solutions can be further developed. We do not claim any of those solutions are final, and this is a discussion of fitness and relevance only.

6.1 Comparison and Measures

Our work was made relative to a ring geometry DHT, but there are many design differences between this work and the existing mature DHTs like Chord, Bamboo and Tapestry. If the solutions is to be used for any of the other geometries the solutions must be adapted, not the underlying DHT.

Here we will try to discuss how the solutions might be adapted for two ring geometry DHTs, and then discuss why they cannot be adapted directly for the other geometries.

6.1.1 Chord

Chord is the base for most of the algorithms in Accord. Both uses a ring geometry and skiplist-like routing, and both uses asynchronous messages for working with a set of well defined protocols. But since Chord and Accord are so similar; are the solutions as valid for Chord as for Accord?

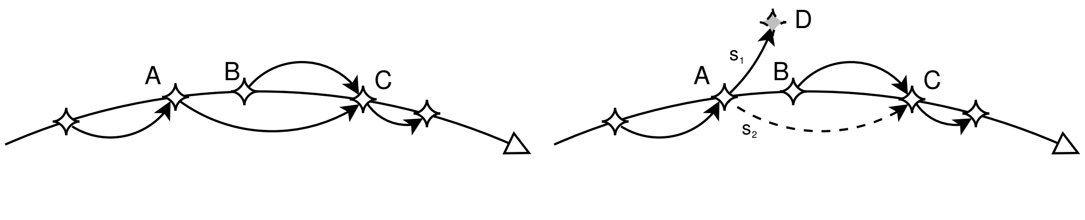
Validity Both Chord and Accord is a ring geometry DHT, although Accord have ≪reversed≫ the ring. So most solutions will work on Chord if predecessors and successors are exchanged. In the case of lookup result validity, one have to make a lookup trace in Chord and Accord and see the similarity.

When a lookup request in Chord (assume recursive lookup) arrives at a host, it is checked for if the node's successor is the index's successor as well, and if so it is the node. In Accord it is always forwarded towards its owner until it's a direct hit. The difference here is that in Chord, the successor link defines the successor's responsibility of indices, and Accord uses the successor to define the local node's responsibility.

This can be further investigated by reversing the Accord ring. This makes it the *predecessor* that defined local ownership, and, like Chord, the successor defines the ownership of it's successor node. If lookups are to be resolved at index's owner, the owner must know its exact responsibility range, and thus having a correct predecessor. This is similar to Chord in every way except two things; responsibility knowledge location (owner node), and lookup replying (from the owner, not its predecessor).

From this it can easily be deducted that the lookup consistency solutions can be adapted from Accord to Chord, and those DHT's that uses the ring geometry.

Predecessor Resolving Join Protocol (PRJP) In Accord, the base of the ≪join protocol≫ is the successor link resolving part (see Section 5.3). If adapted for Chord, this becomes a ≪predecessor-resolving≫ join protocol with the successor routing lookup procedure. The novel solution is to swap all predecessors with successors and visa versa in the algorithm shown in Figure 11 and 12.



(a) Possibly Inconsistent Tables?

(b) Consistent Tables in Chord

For Chord this is example of two situation where consistency can be questioned. Figure 15(a) shows a state where a node has ≪joined≫, but will not receive lookup requests and is not assigned index ownership. Figure 15(b) shows a situation where a node has ≪died≫ in Chord, lookups are consistent, but fails for the dead node's successor.

Figure 15: Consistency in Chord

Figure 15 shows two scenarios in Chord where the routing tables might be incorrect yet consistent, and seemingly correct, but may be inconsistent; similar to the two situations shown in Figure 6. The first situation is the state of a part of a Chord ring after a node (the one outside the ring) has ≪died≫. Since it is difficult to know wether a node is on the ring, as shown in Figure 15(a), Chord needs some protocol to remove this insecurity, which could be the source of its inconsistent lookups.

The same can be argued for Bamboo, Pastry and other DHTs with the ring geometry as bottom line. But tree geometry DHTs are more reliant on what form of distance metric is used, and how this interferes with neighborhood knowledge. If a node in Tapestry doese'nt know any of its two immediate neighbors, it will generate inconsistent lookups, although it can still repair it self by searching for nodes nearby and *backtrack* to the local node. This still does not solve the inconsistency problem.

Since both the *longest prefix* (*XOR*) or *longest suffix* metrics need accurate knowledge about *both* immediate neighbors for guaranteed consistent routing on a local level, the tree geometry needs a different but similar protocol, which in some way guarantees that each neighbor ≪gives away≫ its part of the ownership in a two-part commit protocol. This is not discussed further here, as it requires a thorough analysis of each failure scenario as for the ring model.

Membership The notion of membership is looser in Chord than in Accord, and this has the consequence of putting question to what defines membership in Chord? At least two definitions of membership can be made in Chord, but both have flaws as it is.

1. *Routing:* By defining that being member of the Chord ring is to be part of the cycle[29], we make the mistake by that nodes may after some failure be removed from the *cycle*. There is also the problem that a node is still not a member even after completing join, that waits until its predecessors in the cycle updates its successor link. This is a volatile definition.

2. *Join:* Another way to define membership in Chord is to say that a node is a full member the moment the join protocol has notified its successor of it being there. This is also problematic, as then routing to the newly joined node will fail until its predecessors in the cycle updates its successor link. This is a stable definition, although it may be too stable.

Previous studies in Chord have per definition used definition 1, which does not really reflect the true state of the DHT, but secures a consistent membership state relative to routing. But in order to make membership work ≪faster≫, we propose to use definition 2 and employ a predecessor resolving join protocol (PRJP). This will ensure that membership is activated at time of join, and offer a consistent lookup relevant to that membership definition, and to the consistency definition in Table 3 in Section 3.1.2.

Locking Membership Management Both the pure predecessor resolving join protocol (PJRP) and the Chord join protocol have the problem of concurrent joins. If two nodes try to join with the same successor in Chord, or predecessor in Accord, the consistency of the tables are at stake. In Chord this only makes a short term ≪eviction≫ of one of the joined nodes, which will have to rejoin later.

With the PJRP, four tables at three different nodes will have to be updated correctly in order to achieve consistent states at the three consecutive nodes. Like with databases, this can be solved by locking concurrent access to the join procedures, and to avoid dead-locking a global ordering scheme should be used so that the nodes are locked in *the order of the cycle.*

Lookup Protocol in Chord When nodes leave in Chord, they just form a *hole* in the ring until the node is evicted from its predecessors successor list. One of the problems lies in that the lookup protocol, if unsuccessful with contacting a certain node, retries the lookup with an ≪ignore list≫ ([14], see source code). which includes the unsuccessful node. The new routing will then try the *next* successor on the same stage as the failed lookup result, which may at this stage have returned (or regained network communication). This adds to the inconsistency rating for Chord specifically.

Firstly Chord needs a lookup protocol that does not generate inconsistency with its own lookup protocol, then it needs a consistent leave protocol, that can evict nodes that don't respond to lookup messages as well as tells of activity (like keep-alive protocols).

Leaving Problems The argument of Fail-fast nodes are the exact same for Chord as for Accord. If a node in Chord looses contact with the rest of the ring for a given time period, the rest of the nodes will start removing it from its routing tables, which may cause inconsistency with regard to routing.

In order to avoid inconsistency, the successor and predecessor should use the leave protocol as proposed in Section 5.3 (*I-Am-Alive*). The commit based leave protocol and the PRJP, Chord should be able to show significantly better consistency ratings. This is a topic for further study, and is not discussed further in this thesis.

Another problem that comes from nodes returning to the ring before they have been completely evicted from the routing tables. In order to avoid this, we propose to make each nodes *fail-fast*. That means they must monitor their own successfull network activity with the ring, and simply shut down if there has been a too long period of inactivity. This was discussed in Section 4.2.1, and will not be discussed further here.

The tree geometry leave protocol may need special adaptations for guaranteeing that the leaving node is not assigned the owner of any index assigned to its neighboring node from the leave and onward. This can be easily done since each routing table change moves a part of the indices from the leaving node to its neighbor on each side. The properties of this depends on the DHT's distance metric, and must be analyzed individually.

6.1.2 Other Ring geometry DHTs

Now we know there are measures that can work for Chord, but what about the other ring geometry DHTs? All the proposals in this thesis (discussion) on consistency are shown to work for all ring geometry DHTs that roughly resemble Chord. There is no need for other properties than the successor (or predecessor) based routing, so DHTs like Bamboo and Pastry will be able to benefit as much.

6.1.3 Non-ring DHTs

A question then is; can it work for the other geometries. Not without modifications; because many of the proposals are dependent on the successive neighbors and global ordering as used in routing. Tree geometry DHTs can use various adaptations of the protocols, and definitely use the non-protocol specific means proposed in this thesis like fail-fast nodes.

6.2 Other Observations made in the Thesis

During the project observations not related to consistency have also been made. These observations are not thoroughly studies or analyzed, although tested for usability and correctness.

6.2.1 Dynamic Routing Table Preferences

In the course of the Accord project, it was assumed that the system should not be needed to configure the actual sizes of the routing tables. In order to counter the need for knowledge about a systems scale, we developed some rules (see Section 5.3.5 and Table 7) to approximate what is optimally needed as the sizes of these tables. Considerations of optimality quantities were not made, although these were put in as configurable values.

min_succ Minimum number of successors. Before any fingers are resolved, ensure that this number of consecutive successors are correctly in the routing tables.

succ_ratio Above the number of successors and fingers of *min_succ*, how many of the additional table entries are to be successors (as opposed to fingers). This can be used to make an increase in the number of successors above *min_succ* as the number of fingers grows.

What we found was sound ≪stabilization rules≫ that can determine a set of successors and fingers to optimize the stability vs. efficiency runtime for each node separately.

The rules are not studies beyond the notion of mathematically correctness, and the effect of using such rules in a DHT during churn has not been analyzed, simulated nor tested and measured. The amount of messages generated by this protocol in addition to *normal* network usage is minimal except in the ≪minimal≫ state where the number of successors does not reach *min_succ*.

The rules are only tested to the degree that they work; e.g. they produce the tables desired for a given state in the DHT.

6.2.2 Iteration Restricting Arguments in Lookup/Routing

It has also been observed that it is simple to combine iterative and recursive lookup in a single routing request (see Section 5.2). The studies of lookup during churn have been studying solely the differences between pure iterative and pure recursive lookup, which *could* be not the best options in the world of lookup algorithms.

6.3 Conclusion

In this thesis we have shown that active and consistent membership management is a possible way of combating inconsistent lookup. With the algorithms and means shown here related to consistency built into a ring geometry DHT, consistency should be reduced significantly. Not all problems with these protocols have been thoroughly studied, which is a work left for further study.

The case for the protocols show they work with simple testing, and to produce consistent tables both *before* and *after* node join and leave, and analysis show they produce consistent lookups even *during* the join process. These protocols are mathematically described through state machines, and *should* be possible to prove consistent in *all* failure scenarios, but this is left for further study.

6.4 Further Works

As the protocols are not actually tested in a live DHT during churn, nor simulated with the DHT simulator bundled with Chord[14], this is a job for further study directly linked to this work. Testing should be done with an implementation of the protocols in a mature DHT to be able to compare results with no-membership-management results for consistency rating, CPU and network cost.

Two candidates for this testing are Bamboo, a tree/ring hybrid geometry DHT, and Chord, a pure ring DHT. Bamboo has the advantage of being a Java implemented DHT, giving the possibility of easier to port. Disadvandages with Bamboo is the network event interface, giving disadvantage to both locking and waiting protocols.

Chord, the other candidate is thread-based, like Accord, but programmed in C++. Chord is also no longer in active development, which makes it a stable platform on which to implement the protocols. But when the Accord project was started, it was partly because of a set of unnecessary dependencies in Chord combined with a not-so-good to missing documentation. Chord is documented good enough to develop *for* Chord, but not enough to develop protocols *in* Chord without a long pre-study of its inner workings.

The effects of the non-protocol means of reducing inconsistency should also be tested or simulated with a live DHT in Churn. Since none of these are dependent on a ring geometry DHT, and often require less modification of the original DHT they should be easier to add to the DHT, and to test or simulate.

The effect mostly interesting with regard to the different means is their effect on; consistency in lookup, lookup latency, network throughput per node and CPU use (in case it increases the computational work of nodes significantly).

Dynamic Table Settings The *stabilization* protocols developed for Accord should be possible to simulate and test further to be able to choose optimal settings for any size DHT. This protocol could also be adapted to find optimal settings *not* related to routing tables and their sizes. In the case for tree geometry DHTs, it must be developed a new scale estimation protocol, which can be used for similar purposes.

A Project Overview

Accord has gone through many versions, from the first version (0.1) in september 2004, to the final (0.9) version. The first version had less than 2000 lines, only 6 classes. In between there were several rebuilds of the system, from design and up to almost complete implementation, until some major design flaw, Java API limitation, and even Java Runtime Error stopped the development.

The final version was a redesign from a February 2005 (0.8) version that incorporated a lot of code from earlier versions, but had a redesign (and reimplementation) of the membership protocols. The total code lines in Accord amounts to approximately 4400, then including the *AccordNodeTest* class, used for bootstrapping Accord system, a GNU Make–file, a statistics–calculating script, and a clear–text config file.

Accord consists of five packages each with a set of classes to do some job. Each package and class is documented with *JavaDoc* and is compiled with both *JavaDoc* to `HTML`, and with *Doxygen* to LaTeX(then compiled to DVI and converted to PostScript and PDF).

`net.morimekta.accord` *(Accord Core Services)*

The main services of Accord, following the rules outlined in Section 5.3. The four main classes are; the *Stabilizer*, following the rules outlined in Section 5.3.5; the *IAmAlive* service, which employs the *keep–alive* protocol for Accord; the *Membership* service, which implements the ≪join≫ and ≪leave≫ protocols; and the *Lookup* service, managing lookup requests for many of the other services, and for other users of Accord.

`net.morimekta.accord.tables` *(Accord Routing Tables)*

This package contains the three routing tables of Accord; the *FingerTable*, the *SuccList* and the *PredList*. In addition it holds a generalization of the routing tables called *OverlayContainer*, and a collected class combining the three tables into a complete routing table system called *LookupTable*.

`net.morimekta.net` *(Asynchronous Network Communication)*

On order to achieve asynchronous communication between the nodes in Accord, we built an UDP[36] based communication protocol. Passing around *Message* instances through a *Message-Socket*, and possibly invoking a *MessageService*. And since nodes in Accord are indexed, a *Location* class (inheriting *InetSocketAddress*) with indexing were built.

`net.morimekta.util.index` *(Index Utilities)*

Index utilities that enables arithmetics and dynamic creation and comparison of indices. The class *Index* is the index itself with operations for adding, substraction and comparison. The class *IndexFactory* is an abstract factory class for creating indices from data or encoded strings. *SHA1Factory* is a specialized factory for producing SHA1 hashed indices.

`net.morimekta.util.std` *(Standard Utilities)*

This package contains four classes that that does arbitrary jobs for Accord, like; the class *Log* is an active logger that does formatted timestamps down to the correct millisecond; the class *Options* parses and runs an option string with or without argument through an abstract method `parse(c,arg)`; the class *Config* loads key–value pairs from a config file and stores it into an object (or static class); and the class *STD* that contains small and simple methods for unsigned big–integer arithmetics and various byte–string operations.

Table 8 holds a list of the classes grouped by package with its code line count, line count and byte size.

Package and Class	***Code***[1]	***Lines***[2]	***Bytes***
`net.morimekta.accord.AccordNode`	140	298	8959
`net.morimekta.accord.Conf`	27	51	1641
`net.morimekta.accord.IAmAlive`	253	367	13277
`net.morimekta.accord.Lookup`	462	806	32929
`net.morimekta.accord.Membership`	972	1394	61961
`net.morimekta.accord.Stabilizer`	309	537	22696
`net.morimekta.accord.tables.OverlayContainer`	185	444	11830
`net.morimekta.accord.tables.PredList`	131	179	7096
`net.morimekta.accord.tables.SuccList`	128	177	6822
`net.morimekta.accord.tables.FingerTable`	126	178	6378
`net.morimekta.accord.tables.LookupTable`	137	275	9051
`net.morimekta.net.Location`	128	299	8977
`net.morimekta.net.Message`	137	411	10992
`net.morimekta.net.MessageService`	5	23	684
`net.morimekta.net.MessageSocket`	307	627	19876
`net.morimekta.util.index.Index`	75	265	7280
`net.morimekta.util.index.IndexFactory`	18	62	1407
`net.morimekta.util.index.SHA1Factory`	53	95	2455
`net.morimekta.util.std.Config`	167	303	13128
`net.morimekta.util.std.Log`	80	184	4669
`net.morimekta.util.std.Options`	99	184	5287
`net.morimekta.util.std.STD`	243	583	17748
SUM	4182	7742	275143

1. Code lines are calculated with the UNIX command:
   ```
   % grep -c "[;{}]" $(find -name *.java | sort)
   ```
   from the source code root folder for Accord. This is then a number close to the actual lines of *Java code* in the file.
2. This is the number of *newline characters* in the source file.

The Accord Library (version 0.9) consists of 22 Java classes shown in this table distributed over five packages. Each class is documented with *JavaDoc*, authored by Stein Eldar Johnsen. The exception is a bundled library class for encoding and decoding BASE64 strings which is copyrighted Robert Harder, see http://iharder.net/base64 for more info about the class and its properties and development.
There is also a test-node built for live-testing of the DHT overlay, called `AccordNodeTest`, in a package `test`, which is mainly just a bootstrapper for an Accord node with options and some lookup functionality and statistical testing.

Table 8: Class Overview

B Enclosed CD–ROM Overview

With this thesis a CD–ROM is enclosed including the source code of the Accord Library, its documentation and the thesis document.

`/Accord/src/*` The Accord source code, scripts and default config files.

`/Accord/doc/*` Accord Documentation Overview in JavaDoc HTML format.

`/Accord/doc/doxygen/*` Accord documentation compiled with *Doxygen.* With PostScript and PDF format files (`accord.ps` etc.).

`/Thesis/*` Thesis document in PostScript and PDF file format.

Of the content on the CD–ROM only the thesis document is printed.

Doxygen is a documentation format like JavaDoc, but understands C, C++, Java and more languages, and can generate HTML (like JavaDoc), XML Help files, MAN pages (*NIX style) and generate various diagrams describing the code and its layout.

References

[1] *Gnutella.com.* Web page. URL: http://www.gnutella.org/.

[2] *Jabber: Open Instant Messaging and a Whole Lot More, Powered by XMPP.* Web page. URL: http://www.jabber.org/.

[3] *The Bamboo Distributed Hash Table.* Web page. URL: http://bamboo-dht.org/.

[4] *Napster - All the music you want. Any way you want it.* Web page. URL: http://www.napster.com/.

[5] *FindLaw Legal News: Special Coverage: Napster.* Web page and article collection. URL: http://news.findlaw.com/legalnews/lit/napster/.

[6] *Kazaa.* Web page. URL: http://www.kazaa.com/.

[7] Robert Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO '93)*, October 1993.

[8] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*, pages 311–320, June 1997.

[9] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California at Berkeley, Berkeley, CA, USA, April 2001.

[10] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing (STOC '97)*, pages 654–663, May 1997.

[11] John P Hayes and Trevor Mudge and Quentin F Stout and Stephen Colley and John Palmer. A microprocessor-based hypercube supercomputer. *IEEE Micro*, 6(5):6–17, 1986.

[12] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up Data in P2P Systems. *Communications of the ACM, Vol 46, No. 2*, February 2003.

[13] Sylvia Ratnasamy, Paul Francis, Mark Handley, and Richard Karp. A scalable content-addressable network. In *Proceedings of ACM conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*, August 2001.

[14] *The Chord/DHash Project.* Web page. URL: http://pdos.csail.mit.edu/chord/.

[15] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *Proceedings of ACM conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*, August 2001.

[16] Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[17] G. 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).*, March 2003.

[18] Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[19] Ben Y. Zhao, Yitao Duan, Ling Huang, Anthony D. Joseph, and John D. Kubiatowicz. Brocade: Landmark Routing on Overlay Networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[20] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[21] Shelley Zhuang, Dennis Geels, Ion Stoica, and Randy Katz. On Failure Detection Algorithms in Overlay Networks. In *IEEE Community Society 24th Annual Conference (INFOCOM '05)*, March 2005.

[22] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoika. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of ACM - Data Communications Festival (SIGCOMM '03)*, August 2003.

[23] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowizc. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[24] Anthony D. Joseph Ben Y. Zhao and John D. Kubiatowicz. Locality-aware mechanisms for large-scale networks. In *Workshop on Future Directions in Distributed Computing (FuDiCo)*, June 2002.

[25] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Proximity neighbor selection in tree-based structured peer-to-peer overlays. Technical Report MSR-TR-2003-52, Microsoft Research, Redmond, WA, USA, August 2003.

[26] Byung-Gon Chun, Ben Y. Zhao, and John D. Kubiatowicz. Impact of Neighbor Selection on Performance and Resilience of Structured P2P Networks. In *Proceedings of the 4th International Symposium on Peer-to-Peer Systems (IPTPS '05)*, February 2005.

[27] F. Dabek, J. Li, E. Sit, J. Robertson, M. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[28] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[29] David Liben-Nowell, Hari Balakrishnan, and David Karger. Observations on the dynamic evolution of peer-to-peer networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[30] Supriya Krishnamurthy, Sameh E-Ansary, Eirik Aurell, and Seif Haridi. A Statistical Theory of Chord under Churn. In *Proceedings of the 4th International Symposium on Peer-to-Peer Systems (IPTPS '05)*, February 2005.

[31] J. Li, J. Stribling, T. Gil, R. Morris, and F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proceedings of the 3rd International Symposium on Peer-to-Peer Systems (IPTPS '04)*, February 2004.

[32] J. N. Grey, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency on shared data bases. In *Proceedings of the 1st International Conference on Very Large Databases*, pages 25–33, 1975.

[33] M. Tamer Öszu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 2nd edition, 1999. ISBN 0-13-659707-6.

[34] P. Jones, Motorola, and Cisco Systems. Rfc 3174: Us secure hash algorithm 1 (sha1), September 2001. URL: http://www.ietf.org/rfc/rfc3174.txt.

[35] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. Efficient routing for Peer-to-Peer Overlays. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[36] J. Postel. Rfc 768: User datagram protocol, August 1980. URL: http://www.ietf.org/rfc/rfc768.txt.