Chord: A scalable peer-to-peer lookup service for Internet applications

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan[*]

Abstract

Efficiently determining the node that stores a data item in a distributed network is an important and challenging problem. This paper describes the motivation and design of the *Chord* system, a decentralized lookup service that stores key/value pairs for such networks. The Chord protocol takes as input an m-bit identifier (derived by hashing a higher-level application-specific key), and returns the node that stores the value corresponding to the key. Each Chord node is identified by an m-bit identifier and each node stores the key identifiers in the system closest to the node's identifier. Each node maintains an m-entry routing table that allows it to look up keys efficiently. Results from theoretical analysis, simulations, and experiments show that Chord is incrementally scalable, with insertion and lookup costs scaling logarithmically with the number of Chord nodes.

1 Introduction

A review of the features included in recent peer-to-peer applications yields a long list. These include redundant storage, permanence, efficient data location, selection of nearby servers, anonymity, search, authentication, and hierarchical naming. At their core, however, all these applications need an efficient method for determining the location of a data item. The contribution of this paper is a protocol that solves the lookup problem and a simple system that uses it for storing information.

The Chord system is an efficient distributed lookup service based on the Chord protocol. The Chord system supports five operations: the addition and departure of Chord server nodes, and insert, update, and lookup of unstructured key/value pairs. All operations use the lookup primitive offered by the Chord protocol. We have used the Chord system to build a peer-to-peer file sharing application [2].

The Chord protocol supports just one operation: given a key, it will determine the node responsible for storing the key's value. The Chord protocol uses a variant of consistent hashing [11] to assign keys to Chord server nodes. Under consistent hashing load tends to be balanced (all nodes receive at most $(1 + \epsilon)$ times the average number of key/value pairs). Also when an N^{th} node joins (or leaves) the network, on average only an $O(1/N)$ fraction of the key/value pairs are moved to a different location.

Previous work on consistent hashing assumed that nodes were aware of most other nodes in the network, making it impractical to scale to large number of nodes. We show how each node can get by with "routing" information about a small number of other nodes. Because the routing table is distributed, a node resolves the hash function by communicating with a few other nodes. In the steady state, in an N-node network, each node maintains information only about $O(\log N)$ other nodes, and resolves all lookups via $O(\log N)$ messages to other nodes. We also explain how the hash function and the routing information are revised when a node joins or leaves the network. These updates require $O(\log^2 N)$ messages when a node joins or leaves.

[*] Authors in reverse alpha-betical order.

The final contribution of this paper is an evaluation of the Chord protocol and system. We present proofs that support the theoretical claims. We also present simulation results with up to 10,000 nodes that confirm that the theoretical results are obtainable in practice. Finally, we present measurements of an efficient implementation of the Chord system. These measurements confirm the simulation results.

The rest of this paper is structured as follows. Section 2 contrasts Chord with related work. Section 3 presents the system model that motivates the Chord protocol. Section 4 presents the base Chord protocol. Section 5 presents extensions to handle concurrent joins and failures. Section 6 outlines results from theoretical analysis of the Chord protocol. Section 7 confirms the analysis through simulation. Section 8 presents the implementation of the Chord system. Finally, Section 9 summarizes our conclusions.

2 Related Work

Conceptually the Chord system functions analogously to the DNS system [16]. Both systems map names to values. Chord's algorithms have no special servers, however, in contrast to DNS which relies on a set of special root servers. In addition, Chord doesn't put restrictions on the format and meaning of names; Chord names are just the keys of key/value pairs. Chord doesn't attempt to solve the administrative problems of DNS.

The Chord system may also be compared to to Freenet [5, 6]. Like Freenet, Chord is decentralized, symmetric, and automatically adapts when hosts leave and join. Unlike Freenet, Chord queries always result in success or definitive failure. Furthermore, Chord is scalable: the cost of inserting and retrieving values, as well as the cost of adding and removing hosts, grows slowly with the total number of hosts and key/value pairs. Chord's gains come at the cost of anonymity, which is implemented separately [2].

The Ohaha system [19] uses a consistent hashing-like algorithm for mapping documents to nodes, and a Freenet-style method for routing queries for documents. As a result, it shares some of the weaknesses of Freenet. Archival Intermemory uses an off-line computed tree to map logical addresses to machines that store the data [4].

The Globe system [3] has a wide-area location service to map object identifiers to locations to support moving objects. Globe arranges the Internet as a hierarchy of geographical, topological, or administrative domains, effectively constructing a static world-wide search tree, much like DNS. Information about an object is stored in particular leaf domain and pointer caches provide search short cuts [22]. As pointed out by the authors, the search tree doesn't scale, because higher-level nodes in the tree serve large number of requests and also have high storage demands.

The distributed data location protocol developed by Plaxton *et al.* [7], a variant of which is used in OceanStore [12], is perhaps the closest algorithm to the Chord protocol. It provides stronger guarantees than Chord: like Chord it guarantees that queries make a logarithmic number hops and that keys are well balanced, but the Plaxton protocol also ensures, subject to assumptions about network structures, that queries never travel further in network distance than the node where the key is stored. Chord instead has a heuristic to achieve network proximity and its protocols are substantially less complicated.

Chord's routing procedure may be thought of as a one-dimensional analogue of the Grid [14] location system. The Grid relies on geographic-location information to route its queries, while Chord doesn't require the availability of geographic-location information.

Chord can be used as a lookup service to implement a variety of systems, as discussed in Section 3. In particular, it can help avoid single points of failure or control that systems like Napster [18] possess, and the lack of scalability that systems like Gnutella [9] display because of their widespread use of broadcasts.

Function	**Description**
`insert(key, value)`	Inserts a `key/value` binding at r distinct nodes. Under stable conditions, exactly r nodes contain the key/value binding.
`lookup(key)`	Returns the value associated with the key.
`update(key, newval)`	Inserts the `key/newval` binding at r nodes. Under stable conditions, exactly r nodes contain key/newval binding.
`join(n)`	Causes a node to add itself as a server to the Chord system that node n is part of. Returns success or failure.
`leave()`	Leave the Chord system. No return value.

Table 1: API of the Chord system.

3 System model

The Chord protocol has essentially one operation: given a key, it will determine the node responsible for the key. One can construct a wide range of systems using this primitive. To guide the explanation of the protocol this section defines one such a system, which we have labeled the Chord system.

The Chord system provides a distributed lookup service that allows applications to insert, lookup, and delete values using a key as a handle. The Chord system treats the key simply as an array of bytes and uses it to derive a unique, effectively random m-bit *key identifier*, and does not associate any meaning to the key provided by the application. Like a key, the value provides by the application is simply treated as an array of bytes. Depending on the application, these values could correspond to network locations where application data or services may be found (in which case the Chord system helps in the "rendezvous" process), or to the actual data itself (e.g., files). We expect the predominant use of the Chord system to be as a lookup service for rendezvous, rather than for transferring documents or large files.

The API provided by the Chord system consists of five main functions, shown in Table 1. When `insert(key, value)` is called, Chord inserts the key/value pair at r carefully chosen nodes. The quantity r is a Chord system parameter that depends on the degree of redundancy desired. When `lookup(key)` is called, Chord efficiently finds the key/value binding from some node in the system, and returns the value to the caller. Finally, Chord allows updates to a key/value binding, but currently only by the originator of the key. This restriction simplifies the mechanisms required to provide correct update semantics when network partitions heal. The Chord system does not provide an explicit `delete` operation—an application that requires this feature may implement it using `update(key, value)` with a value corresponding to the "delete-operation" that is interpreted by the application as such (this choice is arbitrary and independent of the chord protocol). The final two API calls are functions for nodes to join and leave a Chord system.

The Chord system is implemented as an application-layer overlay network of Chord server nodes. Each node maintains a subset of the key/value pairs, as well as routing table entries that point to a subset of carefully chosen Chord servers. Chord clients may, but are not constrained to, run on the same hosts as Chord server nodes. This distinction is not important to the Chord protocol described in this paper.

The service model provided by the Chord system may be thought of as a "best-effort persistence" model. As long as at least one of the r nodes in the Chord network storing a key is available, the key/value binding is persistent. If the underlying network connecting Chord servers suffers a partition, the servers in each partition communicate with each other to reorganize the overlay within the partition, assuring that there will be eventually r distinct nodes storing each binding. When partitions heal, a *stabilization protocol* assures that there will be exactly r distributed locations for any binding in any connected partition. The Chord system does not provide tight bounds on consistency, preferring instead (in the "best-effort" sense) to rely on eventual consistency of key/value bindings. Insertions and updates are also not guaranteed to be atomic.

The Chord system's simple API and service model make it useful to a range of Internet applications, particularly because a wide variety of namespaces and values can be used by a Chord application. For example, to implement lookup functionality for the Domain Name System (DNS), the values stored in the Chord system could correspond to the various DNS records associated with the name. The Chord system can also be used by resource discovery servers storing bindings between networked services (names) and their locations (values) [1, 10, 21, 23].

Today, each application requiring the ability to store and retrieve key/value bindings has to re-implement this basic functionality, often having to reconcile several conflicting goals. For example, a key requirement for DNS is scalability, for which it uses administrative hierarchies and aggressive caching; unfortunately, its caching model, based on a time-to-live field, conflicts with its ability to support rapid updates. Some of today's peer-to-peer file sharing systems show that scalability is hard to achieve; Napster, for example, uses a centralized directory that is a single point of failure; Gnutella relies on broadcasts of increasing scope; and Freenet aggressively replicates documents, but cannot guarantee the retrieval of a document within a bounded number of steps nor update documents. The Chord system can serve as a useful lookup service for these applications.

Based on the needs of applications like the ones mentioned above and conditions on the Internet, we set the following design goals for the Chord system:

1. **Scalability.** The system should scale well to potentially billions of keys, stored on hundreds or millions of nodes. This implies that any operations that are substantially larger-than-logarithmic in the number of keys are likely to be impractical. Furthermore, any operations that require contacting (or simply keeping track of) a large number of server nodes are also impractical.

2. **Availability.** Ideally, the lookup service should be able to function despite network partitions and node failures. While guaranteeing correct service across all patterns of network partitions and node failures is difficult, we provide a "best-effort" availability guarantee based on access to at least one of r reachable replica nodes.

3. **Load-balanced operation.** If resource usage is evenly distributed among the machines in the system, it becomes easier to provision the service and avoid the problem of high peak load swamping a subset of the servers. Chord takes a step in this direction by distributing the keys and their values evenly among the machines in the system. More refined load balancing, for example to deal with a single highly popular key by replicating it, can be layered atop the basic system.

4. **Dynamism.** In a large distributed system, it is the common case that nodes join and leave, and the Chord system needs to handle these situations without any "downtime" in its service or massive reorganization of its key/value bindings to other nodes.

5. **Updatability.** Key/value bindings in many applications are not static; it should be possible for these to be updated by the application.

6. **Locating according to "proximity".** If the target of a query is near the originating node, then the originating node should not have to contact distant nodes to resolve the query. We do not provide formal guarantees for this property, but describe some heuristic modifications that should perform well in practice.

The Chord system could provide other properties as well and, in fact, for certain peer-to-peer application it should. For example, certain applications might require that the system provide anonymity, that inserts be authenticated, that stronger consistency be provided in the face of network partitions, or that the system protect against malicious servers (e.g., ones that lie about their identity). We are optimistic that the protocols we propose can be extended to provide support for these features, but that is beyond the scope of this paper. Instead, this paper focuses on the Chord protocol, which solves the problem of determining the node in a distributed system that stores the value for a given key. This problem is challenging, independent of whether the system offers a simple or a more richer service model.

4 The base Chord protocol

Chord servers implement the Chord protocol, using it to return the locations of keys, to help new nodes bootstrap, and to reorganize the overlay network of server nodes when nodes leave the system. We describe the base protocol in this section for the sequential case, when no concurrent joins or leaves occur and no nodes fail. Section 5 describes enhancements to the base protocol to handle concurrent joins and leaves, and node failures.

4.1 Overview

At its heart, chord provides fast distributed computation of a hash function mapping keys to machines responsible for them. We use a previously developed *consistent hash function* [11, 13], which has several good properties. With high probability[1] the hash function balances load (all machines receive at most $(1+\epsilon)$ times the average number of keys). Also with high probability, when an N^{th} machine joins (or leaves) the network, only an $O(1/N)$ fraction of the keys are moved to a different location—this is clearly the minimum necessary to maintain a balanced load.

The previous work on consistent hashing assumed that most machines were aware of most other machines in the network. This assumption does not scale. We show how each machine can get by with only a small amount of "routing" information about other machines. Because the resolution information is distributed, a machine resolves the hash function by communicating with a few other machines. We describe the information that each machine maintains in the steady state, and the routing process used to resolve the hash function. More precisely, in an N-machine network, each machine maintains information only about $O(\log N)$ other machines, and resolves all lookups via $O(\log N)$ messages to other machines.

Finally, we also explain how the hash function and the routing information are revised when a machine joins or leaves the network. These updates require $O(\log^2 N)$ messages when a machine joins or leaves.

4.2 The Hash Function

The consistent hash function begins by assigning to each node and key in the system an m-bit *identifier*. The identifiers are generated using a base hash function such as SHA-1. The node identifiers are chosen by hashing the IP address (or some other unique ID) of the node to the m-bit identifier space. Similarly, the identifiers of the keys are produced by hashing the keys to the m-bit space. (We will use the term "key" to refer to both the original key and its image under the hash function, as the meaning will be clear from context. Similarly, the term node will refer to both the node and its identifier under the hash function.)

As with any hash function, there is a small chance of a collision where two nodes hash to the same identifier; we take m large enough to make this probability negligible. Alternatively, we can append a unique suffix (such as the node's IP address) to the identifier for each node to ensure unique node identifiers (this has no significant impact on our claimed performance). Colliding identifiers for keys are unimportant as the keys themselves, not just the identifiers, are used to resolve lookups.

[1] High probability does not refer to any distribution assumptions about the input (machines and keys). Rather, our algorithm uses a small random seed to define the hash function and routing scheme. With high probability *in this choice of random seed*, the properties we claim will hold *regardless* of the configuration of machines and inputs.

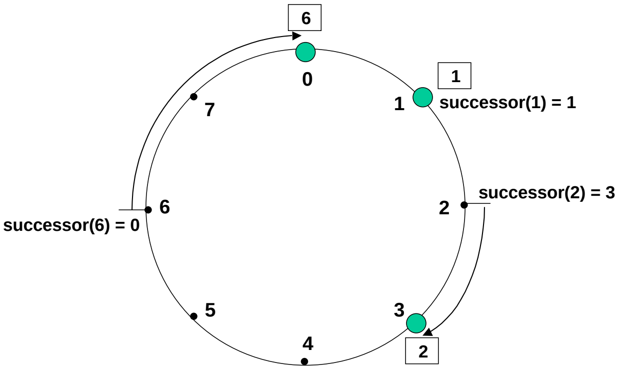


Figure 1: A network consisting of three nodes 0, 1, and 3, which stores three keys 2, 4, and 6. The size of the key-space, m, in this example is 3 bits. Each key (and its associated value) is stored at the successor node of the key. The successor node for an identifier, id, is the first node with an identifier that is equal to or follows id in the clockwise direction on the identifier circle.

Given the identifiers, keys are assigned to nodes in a straightforward way. Each key, k, is stored on the first node whose identifier, id, is equal to or follows k in the identifier space. This node is called the *successor node* of key k, and it is denoted by *successor*(k). If node and key identifiers are represented on a circle marked with numbers from 0 to 2^m, then $successor(k)$ is the first node that we encounter when moving in the clockwise direction starting from k. We call this circle the *identifier circle.*

Figure 1 shows a simple example of a Chord network consisting of three nodes whose identifiers are 0, 1, and 3. The set of keys (or more precisely, keys' identifiers) is $\{1, 2, 6\}$, and these need to be stored at the three nodes. Because the successor of key 1 among the nodes in the network is node 1, key 1 is stored at node 1. Similarly, the successor of key 2 is 3, the first node found moving clockwise from 2 on the identifier circle. For key 6, the successor (node 0) is found by wrapping around the circle, so key 6 is stored at node 0.

Consistent hashing was designed to let nodes enter and leave the network with minimal disruption. To maintain the consistent hashing mapping when a node n joins the network, certain keys previously assigned to n's successor become assigned to n. When node n leaves the network, all of its assigned keys are reassigned to n's successor. No other changes in assignment of keys to nodes need occur. In the example above, if a node were to join with identifier 6, it would capture the key with identifier 6 from the node with identifier 7.

The following results are proven in the paper that introduced consistent hashing [11]:

Theorem 1 *For any set of N nodes and K keys, with high probability:*

1. *Each machine is responsible for at most $(1 + \epsilon)K/N$ keys*
2. *When an $(N + 1)^{st}$ machine joins or leaves the network, $O(K/N)$ keys are moved (and only to or from the joining or leaving machine).*

The consistent hashing paper used a "k-universal hash function" to map nodes and keys to identifiers. This function is defined by a random seed, and the "high probability" statement in the theorem refers to the choice of random seed. In practice, any good hash function (such as SHA-1) should be sufficient to achieve the claimed bounds. To achieve the $(1 + \epsilon)K/N$ bound on load with small ϵ, each node actually needs to run $\log N$ "virtual nodes," each with its own hashed identifier [13]. For simplicity, in the remainder of this section we dispense with the assumption of "virtual nodes." In this case, the load on a machine may exceed the average by (at most) an $O(\log N)$ factor with high probability.

Notation	Definition
$\mathit{finger}[k].start$	$(n + 2^{k-1}) \bmod 2^m$, $1 \le k \le m$
$\mathit{finger}[k].interval$	$[\mathit{finger}[k].start, \mathit{finger}[k+1].start)$, if $1 \le k < m$ $[\mathit{finger}[k].start, n)$, if $k = m$
$\mathit{finger}[k].node$	first node whose identifier is equal to or follows $n.\mathit{finger}[k].start$
successor	immediate successor of node n on the identifier circle; $\mathit{successor} = finger[1].node$
predecessor	immediate predecessor of node n on the identifier circle

Table 2: Definition of variables for node n, where n is represented using m bits.

4.3 Scalable key location

Consistent hashing is straightforward to implement (with the same constant-time operations as standard hashing) in a centralized environment where all machines are known. However, such a system does not scale. In this section we show a distributed implementation of the hash function. More precisely, we discuss what routing information each node needs to maintain, and how a routing decision is made by a node when it does not know the successor of the requested key.

As before, let m be the number of bits in the binary representation of key/node identifiers. Each node, n, maintains a routing table with m entries, called the *finger table*. The i^{th} entry in the table at node n contains the identity of the *first* node, s, that succeeds n by at least 2^{i-1} on the identifier circle, i.e., $s = \mathit{successor}(n + 2^{i-1})$, where $1 \le i \le m$ (and all arithmetic is modulo 2^m). We call node s the i^{th} *finger* of node n, and denote it by $n.\mathit{finger}[i].node$ (see Table 2). Note that the first finger of n is its immediate successor on the circle.

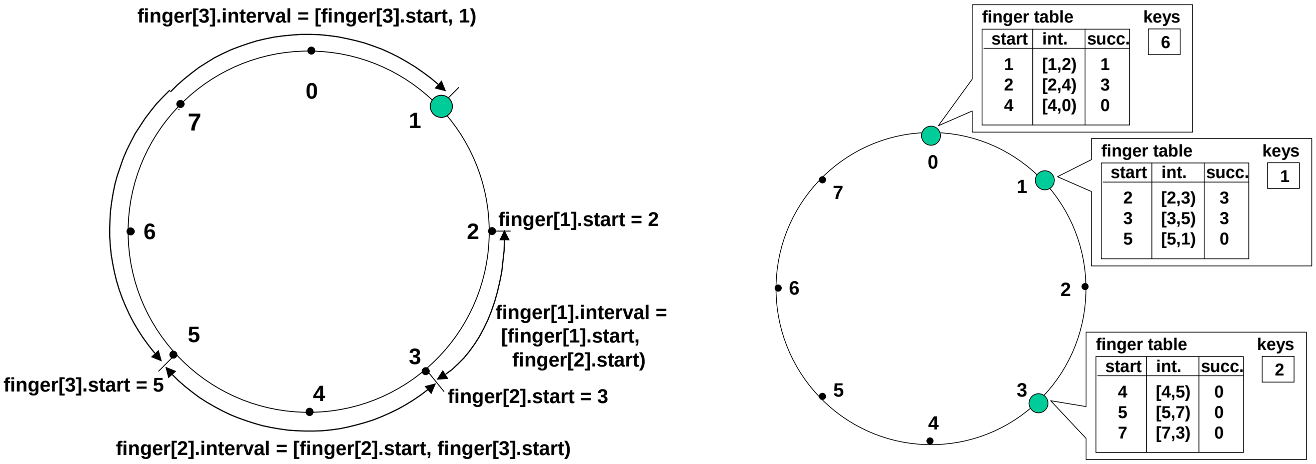
In the example shown in Figure 2, the finger table of node $n = 1$ stores the successors of identifiers $(1+2^0) \bmod 2^3 = 2$, $(1+2^1) \bmod 2^3 = 3$, and $(1+2^2) \bmod 2^3 = 5$, respectively. The successor of identifier 2 is node 3, as this is the first node that follows 2, the successor of identifier 3 is (trivially) node 3, and the successor of 5 is node 0.

It is important to make two observations of this scheme. First, each node stores information about only a small number of other nodes, and the amount of information maintained about other nodes falls off exponentially with the distance in key-space between the two nodes. Second, the finger table of a node may not contain enough information to determine the successor of an arbitrary key k. For example, node 3 in Figure 2 does not know the successor of 1, as 1's successor (node 1) does not appear in node 3's finger table.

What happens when a node n does not know the successor of a key k? To resolve this, node n asks another node in the network to try and find k's successor. Node n aims to find a node closer to k than n, as that node will have more "local information" about the nodes on the circle near k. To accomplish this task, node n searches its finger table for the closest finger preceding k, and forwards the query to that node. As a result the query moves quickly to the target identifier.

To make this search process more precise, we introduce some notations. Consider the i such that $k \in [(n + 2^{i-1}), (n + 2^i)]$. We call this the i^{th} *finger interval* of node n, and denote it by $n.finger[i].interval$ (see Table 2). By definition, the i^{th} finger of n is the first node in n's i^{th} finger interval, if such a node exists. Otherwise, it is the first node following the interval.

The pseudocode that implements the search process is shown in Figure 3. In the pseudocode the notation n.**foo** is used to introduce the function definition for *foo* being executed on node n. To differentiate between remote and local node operations, remote procedure calls and variable references are preceded by the remote node, while local variable references and procedure calls omit the local node. Thus, *n.foo* denotes a remote procedure call on node n, while *foo* denotes a local



(a) (b)

Figure 2: (a) Intervals associated to node $n = 1$, where $m = 1$ (see Table 2). (a) The key and finger tables associated to each node in a network consisting of nodes 0, 1 and 3, respectively, which stores three keys 1, 2, and 6, respectively.

call.

As can been seen in the pseudocode, *find_successor* is implemented by homing in on the immediate predecessor node of the identifier. That node can report that its immediate successor node is also the immediate successor node of the identifier. We implement *find_predecessor* explicitly, because it is used later to implement the join operation (see Section 4.4.1).

The *find_predecessor* function first tests for the case when n is the only node in the network, and therefore its predecessor is the node itself. In this case we simply return node n (in a network with two nodes each node is the predecessor of the other node). The loop terminates when id falls between node n' and its successor, in which case n' is returned as being the id's predecessor. Otherwise, id follows the successor of n', which means that there is at least one finger of n' that precedes id. As a result, *closest_preceding_finger* is called to return the closest finger of n' that precedes id. This value is closer to id than n. Thus, the algorithm always makes progress toward termination at the correct value.

We remark on the correctness of the code. Once we know the predecessor n' of id, the successor of id is simply the successor of n'. This is because we are guaranteed that there is no other node between n' and id; otherwise, that node, and not n', would be the predecessor of id.

A simple optimization for *find_successor* allows it to return early. If we determine that node n is between *finger*$[i]$.*start* and *finger*$[i]$.*node*, we can immediately deduce that *finger*$[i]$.*node* is the immediate successor for id and return that value.

In Section 6, we analyze this algorithm and show the following:

Theorem 2 *With high probability, the number of nodes that must be contacted to resolve a successor query in an N-node network is $O(\log N)$.*

The intuition behind this claim is that each recursive call to *find_successor* halves the distance to the target identifier.

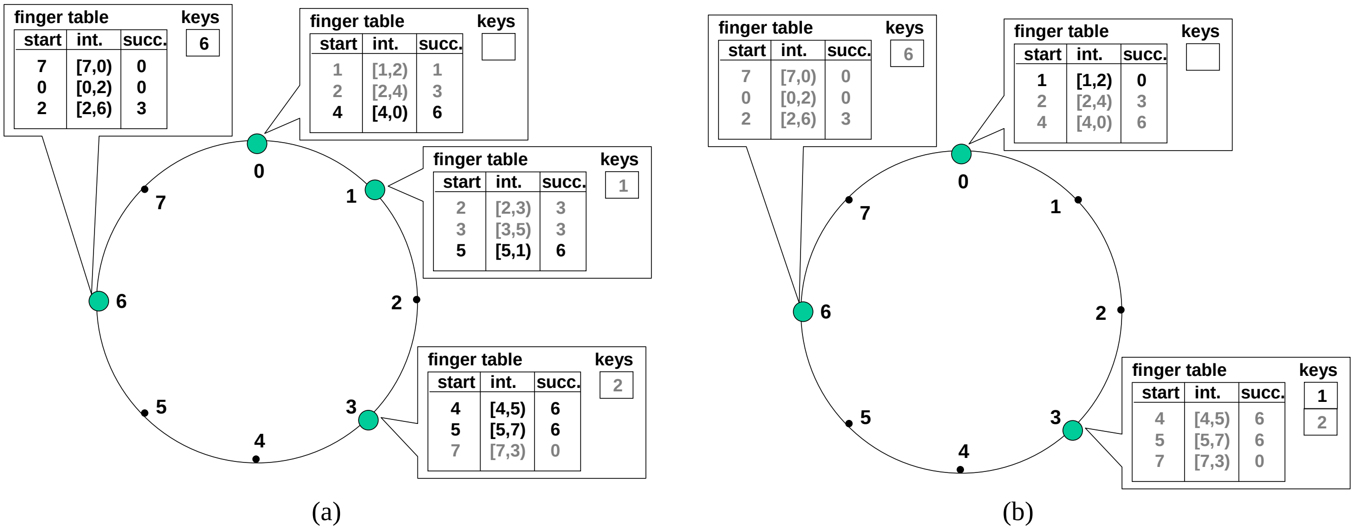
Consider again the example in Figure 2. Suppose node 3 wants to find the successor of identifier 1. Since 1 belongs to the circular interval $[7, 3)$, it belongs to 3.*finger*$[2]$.$interval$; node 3 therefore checks the *second* entry in its finger table, which is 0. Because 0 precedes 1, node 3 will ask node 0 to find the successor of 1. In turn, node 0 will infer from its finger table that 1's successor is the node 1 itself, and return node 1 to node 3.

```
// ask node $n$ to find $id$'s successor
$n$.find_successor($id$)
  $n' = find\_predecessor(id)$;
  return $n'.successor$;

// return closest finger preceding $id$
$n$.closest_preceding_finger($id$)
  for $i = m$ downto $1$
    if (finger[$i$]$.node \in (n, id)$)
      return finger[$i$]$.node$;
  return $n$;
```

```
// ask node $n$ to find $id$'s predecessor
$n$.find_predecessor($id$)
  if ($n == successor$)
    return $n$; // n is the only node in network
  $n' = n$;
  while ($id \notin (n', n'.successor]$)
    $n' = n'.closest\_preceding\_finger(id)$;
  return $n'$;
```

Figure 3: The pseudocode to find the successor node of an identifier id. Remote procedure calls are preceded by the remote node.



(a)

Node 6 — keys: 6

start	int.	succ.
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

Node 0 — keys:

start	int.	succ.
1	[1,2)	1
2	[2,4)	3
4	[4,0)	6

Node 1 — keys: 1

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	6

Node 3 — keys: 2

start	int.	succ.
4	[4,5)	6
5	[5,7)	6
7	[7,3)	0

(b)

Node 6 — keys: 6

start	int.	succ.
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

Node 0 — keys:

start	int.	succ.
1	[1,2)	0
2	[2,4)	3
4	[4,0)	6

Node 3 — keys: 1, 2

start	int.	succ.
4	[4,5)	6
5	[5,7)	6
7	[7,3)	0

Figure 4: (a) The key and the finger tables associated to each node after node 6 joins the network. (b) The key and the finger tables associated to each node after node 3 leaves the network. The changes in the finger tables and of the keys stored by each node as a result of a node joining/leaving are shown in black; the unchanged entries are shown in gray.

4.4 Node joins and departures

In a dynamic network, nodes can join and leave at any time. The main challenge of implementing these operations is preserving the ability to locate every key in the network. To achieve this goal, we need to preserve two invariants:

1. Each node's finger table is correctly filled.

2. Each key k is stored at node $successor(k)$.

It is easy to see that these two invariants will guarantee that $find\_successor$ will be able to successfully locate any key—if a node is not the immediate predecessor of the key, then its finger table will hold a node closer to the key to which the query will be forwarded, until the key's successor node is reached. In the remainder of this section, we assume that these two invariants hold before a node joins or leave the network. We defer the discussion of multiple nodes leaving or/and joining simultaneously to Section 5. Before explaining how joining and leaving are implemented, we summarize the performance of the schemes we are about to define:

Theorem 3 *With high probability, any node joining or leaving an N-node Chord network will use $O(\log^2 N)$ messages to re-establish the Chord routing invariants.*

To simplify the join and leave mechanisms, each node in Chord maintains a *predecessor pointer*. A node's predecessor pointer points at the immediate predecessor of that node, and can be used to walk counterclockwise through nodes on the identifier circle. For clarity, we also introduce a *successor pointer*. The successor pointer points to the same node as $finger[1].node$ (see Table 2).

The rest of this section describes how Chord handles nodes joining and with minimal disruption. (We won't present the implementation of leave, because it is analogous to join.)

4.4.1 Join operation

To preserve the two invariants discussed above, when a node n joins the network, we have to perform three operations:

1. Initialize the predecessor and fingers of node n.

2. Update the fingers and predecessors of existing nodes to reflect the change in the network topology caused by the addition of n.

3. Copy all keys for which node n has became their successor to n.

We assume that the bootstrapping for a new node is handled offline, perhaps by someone configuring the newly joining node n with the identifier of at least one other node n' already in the Chord network. Once this is done, node n uses n' to initialize its state. It performs the above three tasks as follows.

Initializing fingers and predecessor: A straightforward way to learn the predecessor and fingers of node n is to simply ask node n' for them. Figure 5 shows the pseudocode of the *init_finger_table* function that initializes the finger table of node n using this idea. Initializing the predecessor is similar. As an optimization, note that once we learn the i^{th} finger, we check whether this node is also the $(i+1)^{th}$ finger of node n. This happens when *finger*$[i].interval$ does not contain any node, and thus *finger*$[i].node \geq finger[i+1].start$.

As an example, consider the scenario in Figure 4(a), where node 6 joins the network. Assume node 6 knows a node 1, already in the network. Then, node 6 will ask node 1 for the successors of $(6+2^0)$ mod $2^3 = 7$, $(6+2^1)$ mod $2^3 = 0$, and $(6+2^2)$ mod $2^3 = 2$, respectively. In turn, node 1 will return node 0 as being the successor of identifiers 7 and 0, and node 3 as being the successor of 3.

```
// node $n$ joins the network;
// $n'$ is an arbitrary node in the network
$n$.join($n'$)
  if ($n'$)
    init_finger_table($n'$);
    notify();
    $s = successor$; // get successor
    $s.move\_keys(n)$;
  else // no other node in the network to $n$ itself
    for $i = 1$ to $m$
      finger[$i$].$node = n$;
    predecessor $= successor = n$;

// initialize finger table of local node;
// $n'$ is an arbitrary node already in the network
$n$.init_finger_table($n'$)
  finger[1].$node = n'.find\_successor(finger[1].start)$;
  successor $= finger[1].node$;
  for $i = 1$ to $m - 1$
    if (finger[$i + 1$].$start \in [n, finger[i].node)$)
      finger[$i + 1$].$node$ = finger[$i$].$node$;
    else
      finger[$i + 1$].$node$ =
          $n'$.find_successor(finger[$i + 1$].$start$);
```

```
// update finger tables of all nodes for
// which local node, $n$, has became their finger
$n$.notify()
  for $i = 1$ to $m$
    // find closest node $p$ whose $i^{th}$ finger can be n
    $p = find\_predecessor(n - 2^{i-1})$;
    $p.update\_finger\_table(n, i)$;

// if $s$ is $i^{th}$ finger of $n$, update $n$'s finger table with $s$
$n$.update_finger_table($s, i$)
  if ($s \in [n, finger[i].node)$)
    finger[$i$].$node = s$;
    $p = predecessor$; // get first node preceding $n$
    $p.update\_finger\_table(s, i)$;

// if $p$ is new successor of local stored
// key $k$, move $k$ (and its value) to $p$
$n$.move_keys($p$);
  for each key $k$ stored locally
    if ($p \in [d, n)$)
      move $k$ to $p$;
```

Figure 5: The pseudocode of the node joining operation.

Update fingers and predecessors of existing nodes: When a new node, n, joins the network, n may become the finger and/or successor of other nodes in the network. For example, in Figure 4(a), node 6 becomes the third finger of nodes 0 and 1, and the first and the second finger of node 3. To account for this change, we need to update the fingers and successors of the existing nodes, as well. For simplicity, we again limit our discussion to the finger table only.

Figure 5 shows the pseudocode of the $update\_finger\_tables$ function that updates finger tables of the existing nodes. Assume that after n joins the network, n will become the i^{th} finger of a node p. This will happen if and only if (1) p precedes n by at least 2^{i-1}, and (2) the i^{th} finger of node p succeeds n. The first node, p, that can meet these two conditions is the immediate predecessor of $n - 2^{i-1}$. Thus, for a given n, the algorithm starts with the i^{th} finger of node n, and then continues to walk in the counter-clock-wise direction on the identifier circle until it encounters a node whose i^{th} finger precedes n.

Although it might appear that the number of nodes that have their finger tables updated is quite large, this is fortunately not the case. We show in Section 6 that the number of nodes that need to be updated when a node joins the network is only $O(\log N)$ on the average, and with a very high probability is at most $O(\log^2 N)$, where N is the total number of nodes in the network.

Transferring keys: The last operation that has to be performed when a node n joins the network is to move all the keys for which node n has become the successor to n. The pseudocode for this operation, *move_keys*, is shown in Figure 5. The algorithm is based on the observation that node n can become the successor only for keys stored by the node immediately following n. For example, in Figure 4(a), node 6 needs to inspect only the keys stored by node 0. As a result, key 6 is moved to node 6, as node 6 is now the new key's successor.

5 Handling concurrent operations and failures

In practice the Chord system needs to deal with nodes joining the system concurrently and with nodes that fail or leave voluntarily. This section describes modifications to the basic Chord algorithm described in Section 4 to support these situations.

Motivating this section is the observation that a substantially weaker invariant will guarantee the *correctness* of the routing protocol, although time bounds may be sacrificed. As long as every node knows its immediate predecessor and successor, no lookup will stall anywhere except at the node responsible for a key. Any other node will know of at least one node (its successor) that is closer to the key than itself, and will forward the query to that closer node.

5.1 Concurrent joins

The join code in Figure 5 assumes the invariants mentioned in Section 4.4. The invariants may not be true if nodes join the system concurrently. A slightly different version of the code, shown in Figure 6, is required to support concurrent joins. This code focuses on maintaining the correctness of immediate predecessors and successors, since in the worst case more distant relationships can be resolved (though slowly) by nearest-neighbor traversals of the identifier circle.

When node n first starts, it calls $n.join(n')$, where n' is any known Chord node. The $join$ function finds the immediate predecessor and successor of n, notifies those two nodes that they have a new immediate neighbor, and then calls *boot_strap* to fill in n's finger table and initialize n's predecessor.

If multiple nodes with similar identifiers join at the same time, they may all try to notify the same existing predecessor that they are its immediate successor. *notify* ensures that only the newcomer with the lowest identifier will succeed. The others will gradually learn their true immediate neighbors by periodic calls to *stabilize*. *stabilize* periodically checks whether new nodes have inserted themselves between a node and its immediate neighbors. A similar periodic function (not shown in

```
// n joins the network
$n$.join($n'$)
  $s = n'.find\_predecessor(n)$;
  do
    $p = s$;
    $s = p.successor$;
  until $n \in (p, s]$
  $successor = s$;
  $predecessor = p$;
  $p$.notify($n$); // tell p to update its state
  $s$.notify($n$); // tell s to update its state
  $boot\_strap(s)$;

$n$.notify($n'$)
  if ($n' \in (n, successor)$)
    $finger[1].node = successor = n'$;
    $boot\_strap(n')$;
  if ($n' \in (predecessor, n)$)
    $predecessor = n'$;
    $boot\_strap(n')$;
```

```
// let node n send queries to fill in its own tables
$n$.boot_strap($n'$)
  for $i = 1$ to $m$
    $p = n'.find\_successor(finger[i].start)$;
    do
      $s = p$;
      $p = p.predecessor$;
    until ($p < finger[i].start$)
    $finger[i].start = s$;

// verify n's immediate pred/succ
// called periodically
$n$.stabilize()
  $x = predecessor$;
  $x = x.successor$;
  if ($x \in (predecessor, n)$
    predecessor $= x$;
  $x = successor$;
  $x = x.predecessor$;
  if ($x \in (n, successor)$)
    $finger[1].node = successor = x$;
```

Figure 6: Pseudocode for concurrent join. Predecessor functions that parallel their successor equivalents are omitted.

Figure 6) updates the rest of each node's finger table, so that all tables gradually converge on correct values after any joins.

Obtaining the successor by simply calling *find_successor* may not work correctly during the time in which all nodes tables are reacting to a new node. In particular, a node may think that $finger[i].node$ is the first node in an interval, when in fact a new node has joined earlier in that interval. For this reason, we ask the putative successor node whether it does not know a better successor, i.e., whether the predecessor of the putative successor node succeeds $finger[i].node$. This process is repeated until it reaches a successor whose immediate predecessor precedes the key.

Whenever a node notices that it has a new immediate predecessor, it moves appropriate key/value pairs to that predecessor. There are cases in which multiple joins may cause keys to become temporarily inaccessible until sufficient calls to $stabilize$ have been made. This can be solved by serializing the order in which a node accepts new immediate predecessors, which itself is easily achieved by a simple locking protocol between a new node and its immediate successor.

As an optimization, a newly joined node n can ask an immediate neighbor for a copy of its complete finger table and its predecessor. n can use the contents of these tables as hints to help it find the correct values for its own tables, since n's tables will be similar to its neighbors'. Specifically, n's $boot\_strap$ routine can start the query for each table entry at the node referred to by the corresponding entry in its neighbor's table.

5.2 Failures and replication

When a node n fails, nodes whose tables include n must find n's successor (or predecessor), and then n's successor must ensure that it has a copy of the key/value pairs stored in n. In addition, the failure of n must not be allowed to disrupt queries that are in progress as the system is re-stabilizing.

If a node fails, the $stabilize$ procedures of its immediate neighbors will see that it is not responding. The recovery procedure for a node n that notices that its immediate successor has died is as follows. n looks through its finger table for the first live node n'. n then calls $n.find\_successor(n', n)$ (i.e., sends a query to n'), and uses the result as its new immediate successor. A similar strategy works when other table entries are found to be unresponsive.

Fault-tolerant storage of key/value pairs requires replication. To help achieve this, each Chord node maintains a list of its r nearest successors with a simple extension to the code in Figure 6 (in this case, the *successor* scalar variable is replaced by a table). When a node receives an insert, it propagates a copy of the key/value pair to those r successors; it also propagates when it notices that its immediate successor changes during stabilization. After a node fails, queries for its keys automatically end up at its successor, which will have copies of those keys.

After a node failure, but before stabilization has completed, other nodes may attempt to send requests to the failed node as part of a $find\_successor$ lookup. The problem can be detected by timing out the requests, but ideally the lookups would be able to proceed immediately by another path despite the failure. In most cases this is possible: any node with identifier close to the failed node's identifier will have similar routing table entries, and can be used to route requests at a slight extra cost in route length. All that is needed is a list of alternate nodes, easily found in the finger table entries preceding that of the failed node. If the failed node had a very low finger table index, the r successors mentioned above are also available as alternates. Theorem 5 in Section 6 discusses this procedure in more detail. [2]

6 Theoretical analysis

As is discussed in the work on consistent hashing [11], with the proper choice of hash function the identifiers for nodes and keys are effectively random: all analyses can be carried out as if the nodes ended up at N random points on the identifier circle. The same holds true for our analyses, so we will make that assumption.

Theorem 4 *With high probability, the number of nodes that must be contacted to resolve a successor query in an N-node network is $O(\log N)$.*

Proof: Suppose that node n wishes to resolve a query for the successor of k. Let p be the node that immediately precedes the query identifier k. Recall that the query for k eventually reaches p, which returns its immediate successor as the answer to the successor query. We analyze the number of query steps to reach p.

Recall that if $n \neq p$, then n forwards its query to the closest predecessor of k in its finger table. Suppose that node p is in the i^{th} finger interval of node n. Then since this interval is not empty, node n will finger some node f in this interval. The distance (number of identifiers) between n and f is at least 2^{i-1}. But f and p are both in n's i^{th} finger interval, which means the distance between them is at most 2^{i-1}. This means f is closer to p than to n, or equivalently, that the distance from f to p is at most half the distance from n to p.

If the distance between the node handling the query and the predecessor p halves in each step, and is at most 2^m initially, we see that withing m steps the distance will be one, meaning we have arrived at p. In fact, the number of forwardings necessary will be $O(\log N)$ with high probability. After $\log N$ forwardings, the distance between the current query node and the key k will be reduced to $2^m/N$. The expected number of node identifiers landing in a range of this size is 1, and it is $O(\log N)$ with high probability. Thus, even if the remaining steps advance by only one node at a time, they will cross the entire remaining interval and reach key k within another $O(\log N)$ steps. □

The following lemma is the basis of Theorem 3 in Section 4.4, which claims that a node joining the network only needs to send $O(\log^2 N)$ messages to update other nodes' tables.

Lemma 1 *With high probability, every node is a finger (of a given order) of $O(\log^2 N)$ nodes.*

[2] The current implementation takes a slightly different approach to handling failures.

Proof: We begin with an easy expectation argument. Every node has $O(\log N)$ distinct fingers (the easy argument is m fingers, but the $O(\log N)$ bound follows as it did in the previous theorem). Thus the total number of node-successor pairs in the network is $O(N \log N)$. It follows that on average any given node is a finger of $O(\log N)$ nodes.

For a high probability argument, we note that a node n is a finger for n' if $n' + 2^i$ is in the range between n and the predecessor p of n. This happens with probability $(n-p)/2^m$. It is straightforward that with high probability $(n-p) = O(2^m (\log N)/N)$. So for a particular i the probability that a node fingers n is $O((\log N)/N)$, which implies that with high probability $O(\log N)$ nodes finger N at level i. Since there are $O(\log N)$ levels, the total number of nodes fingering n is $O(\log^2 N)$ with high probability. □

We now discuss modifications of the Chord protocol to support fault tolerance. Our focus is not on the loss of data (which can be dealt with by simple replication) but on the loss of routing information. If some of a node's fingers fail, what alternative mechanism does it use to foward successor queries to the appropriate location? To deal with this problem, we modify the Chord protocol to replicate certain routing information. In addition to maintaining its m finger entries, each node also maintains pointers to the first z of its immediate successors on the identifier circle. As will later become clear, z should be large enough that $(1/2)^z$ is very small. Maintaining this information requires only a small constant factor more space on each node. It also involves straightforward modifications to the protocols for joining, leaving, and stabilizing the network which we do not discuss here. We do remark on the change to the routing protocol. If the node to which we want to forward the query (say our i^{th} finger) is down, forward the query instead to the best earlier finger (the $(i-1)^{st}$, or if that is down the $(i-2)^{nd}$, and so on). This sequence should include the z immediate successors.

This replication and routing modification suffices to route around failures. We consider the following model: begin with a network of N nodes with all routing information correct, and suppose that each node fails with probability $1/2$. Eventually the stabilization procedure described in Section 5 will correct the routing information, but in the meantime many of the remaining nodes' tables will refer to failed nodes. The following lemma shows that correct routing still takes place.

Theorem 5 *In a stable network, if every node fails with probability 1/2, then with high probability any successor query returns the closest living successor to the query key.*

Proof: Before the failures, each node was aware of its z immediate successors. The probability that all of these successors fail is $(1/2)^z$, so with high probability every node is aware of its immediate living successor. As was argued in the previous section, if the invariant holds that every node is aware of its immediate successor, then all queries are routed properly, since every node except the immediate predecessor of the query has at least one better node to which it will forward the query. □

In fact, even the efficiency of our routing scheme is preserved in the face of failures.

Theorem 6 *In a stable network, if every node fails with probability 1/2, then the expected time to resolve a query in the failed network is* $O(\log N)$

Proof: We consider the expected time for a query to move from a node that has the key in its i^{th} finger interval to a node that has the key in its $(i-1)^{st}$ finger interval. We show that this expectation is $O(1)$. Summing these expectations over all i, we find that the time to drop from the m^{th} finger interval to the $(m - \log N)^{th}$ finger interval is $O(\log N)$. At this point, as was argued before, only $O(\log N)$ nodes stand between the query node and the true successor, so $O(\log N)$ additional forwarding steps arrive at the successor node.

To see that the expectation is $O(\log N)$ consider the current node n that has the key in its i^{th} finger interval. If n's i^{th} finger s is up, then in one forwarding step we accomplish our goal: the key is in the $(i-1)^{st}$ finger interval of node s. If s is down then, as argued in the previous theorem, n is still able to forward (at least) to *some* node. More precisely, n was aware of z immediate successors; assume $z \geq 2\log N$. If we consider the $(\log N)^{th}$ through $(2\log N)^t h$ successors, the probability that they all fail is $1/N$. So with high probability, node n can forward the query past at least $\log N$ successors. As was implied by Lemma 1, it is unlikely that all $\log N$ of these skipped nodes had the same i^{th} finger. In other words, the node to which n forwards the query has a different i^{th} finger than n did. Thus, independent of the fact that n's i^{th} finger failed, there is a probablity $1/2$ that the next node's i^{th} finger is up.

Thus, the query passes through a series of nodes, where each node has a distinct i^{th} finger (before the failures) each of which is up independently with probability $1/2$ after the failures. Thus, the expected number of times we need to forward the query before finding an i^{th} finger that is up is therefore 2. This proves the claim. □

In fact, our claims hold even if an adversary maliciously chooses an *arbitrary* set of $N/2$ nodes to fail. So long as the adversary is unaware of the specific hash function used to map nodes to the identifier circle, his choice results in the failure of $N/2$ "random" points on the circle, which is precisely what we analyzed above.

7 Simulation Results

In this section, we evaluate the Chord protocol by simulation. We have implemented a packet level simulator that fully provides the functionality of the algorithm described in Sections 4 and 5.

7.1 Protocol implementation and simulator

The Chord protocol can be implemented in a *iterative* or *recursive* style, like the DNS protocol. In the iterative style, a node that is resolving a lookup, initiates all communication: it iteratively queries intermediate nodes for information until it reaches the destination node. In the recursive style, an intermediate node recursively calls the lookup procedure to resolve the query.

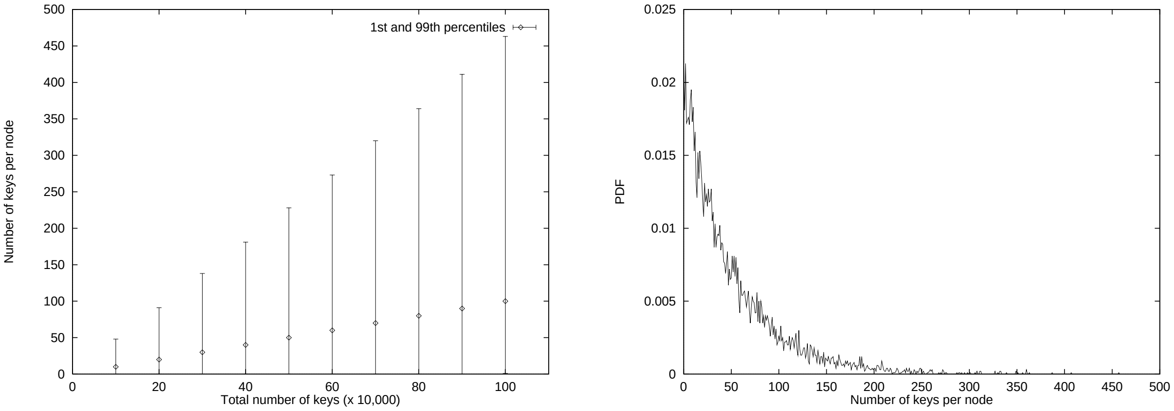
The main advantage of the iterative style is two fold: it is simple to implement (the intermediate nodes just respond to requests, but never initiate communication recursively) and it puts the initiator in control (e.g., it can monitor easily whether a node is responding or not). However, as we discuss in Section 8, there are some disadvantages to the iterative style. The iterative scheme will send queries over long distances repeatedly under certain circumstances. Recursive scheme does a better job of taking short hops when possible. The simulator implements the protocols in an iterative style.

Unless other specified, packet delays are exponentially distributed with the mean of 50 ms. Each node periodically invokes the *stabilization* protocol at an average rate of 0.5 invocations per minute. The time interval between two consecutive invocations by a node is uniformly distributed between 0.5 and 1.5 of the mean value. As shown in [8] in the context of route updates, this choice is likely to eliminate protocol self-synchronization, i.e., all nodes invoking the stabilization protocol at the same time. For key and node identifiers, we use a 24 bit representation. Our implementation assumes that we can transfer any number of keys between two neighbor nodes with only one message. However, we do not expect that this assumption to impact the general behavior of the algorithm as illustrated by the following experiments.

7.2 Load balancing

In this section, we consider the ability of Chord to achieve load balancing. Ideally, given a network with N nodes, and K keys, we would like each node to store N/K keys.

We consider a network consisting of 10^4 nodes, and vary the total number of keys from 10^5 to 10^6 in increments of 10^5. For each value, we repeat the experiment 20 times. Figure 7(a) plots the mean value, the 1st and the 99th percentile of the number of keys per node. The number of keys per node exhibits large variations that increase linearly with the number of keys. For example, in all cases there are nodes that do not store any keys. For a better intuition, Figure 7(b) plots the probability density function (PDF) of the number of keys per node when there are 5×10^5 keys stored in the network. The maximum number of nodes stored by any node in this case is 457, or $9.1\times$ the mean value. For comparison, the 99th percentile is $4.6\times$ the mean value.



(a)

(b)

Figure 7: (a) The mean value, the 1st and the 99th percentiles of the number of keys stored by a node in a 10^4 node network. (b) The probability density function (PDF) of the number of keys per node. The total number of keys is 5×10^5.

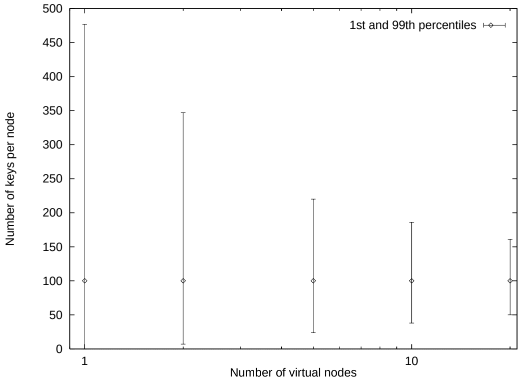
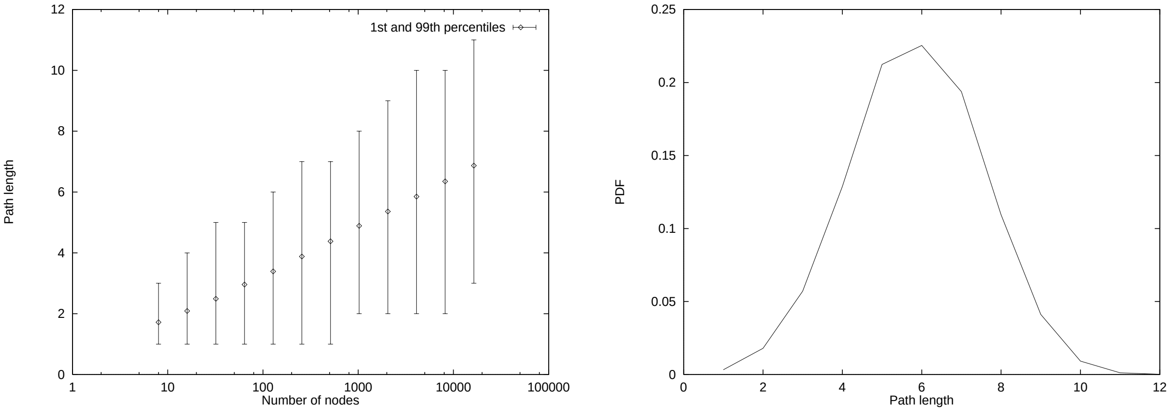


Figure 8: The 1st and the 99th percentiles of the number of keys per node as a function of virtual nodes mapped to a real node. The network has 10^4 real nodes and stores 10^6 keys.

One reason for these variations is that node identifiers do not cover uniformly the entire identifier space. If we divide the identifier space in N equal-sized bins, where N is the number of nodes, the probability that a particular bin does not contain any node is significant. In particular, this probability is $(1 - 1/N)^N$, which for large values of N approaches $e^{-1} = 0.368$.

One possibility to address this problem is to allocate a set of virtual nodes and then map them to real nodes. Intuitively, this will provide a more uniform coverage of the identifier space. For example, if we allocate $\log N$ identifiers to each node, with a high probability each of the N bins contains $O(\log N)$ nodes [17].

To verify this hypothesis, we perform an experiment in which we allocate r virtual nodes to each real node. In this case keys are associated to virtual nodes instead of real nodes. We consider again a network with 10^4 real nodes and 10^6 keys. Figure 8(b) shows the 1st and 99th percentiles for $r = 1, 2, 5, 10$, and 20, respectively. As expected, the 99th percentile decreases, while the 1st percentile increases with the number of virtual nodes, r. In particular, the 99th percentile decreases from $4.8\times$ to $1.6\times$ the mean value, while the 1st percentile increases from 0 to $0.5\times$ the mean value. Thus, adding virtual nodes as an indirection layer can significantly improve load balancing. The tradeoff is that the space usage will increase as each actual node now needs r times as much space to store the information for its virtual nodes. However, we believe that this increase can be easily accommodated in practice. For example, assuming a network with $N = 10^6$ nodes, and



(a)

(b)

Figure 9: (a) The path length as a function of network size. (b) The PDF of the path length in the case of a 2^{12} node network.

assuming $r = \log N$, each node has to maintain a table with $\log^2 N \simeq 400$ entries.

7.3 Path length

One of the main performance parameters of any routing protocol is the length of the path (route) between two arbitrary nodes in the network. Here, we define the path length as the number of nodes traversed by a lookup operation. Recall that the path length is $m + 1$ in the worst case, where m represents the number of bits in the binary representation of the identifiers, and $O(\log N)$ in the average case, where N is the number of nodes. For simplicity, here we assume that there are no virtual nodes.

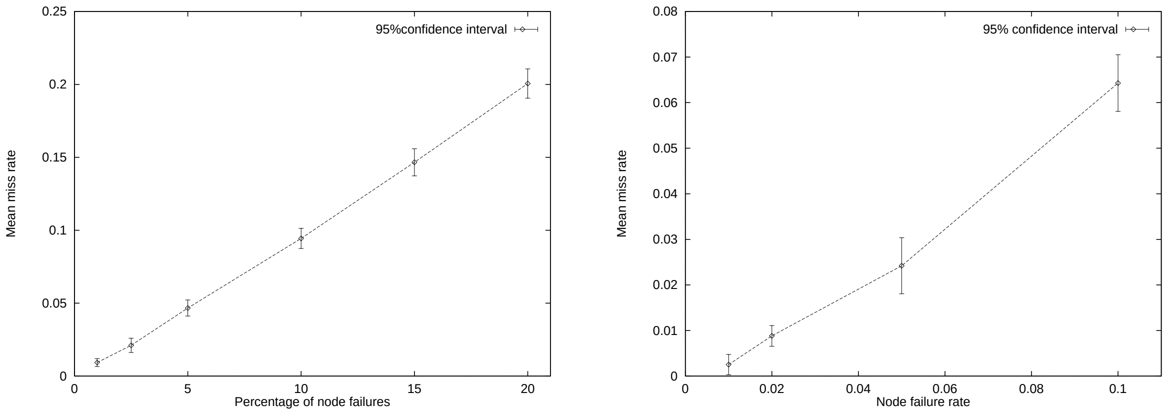
We consider a network with 2^k nodes that stores 100×2^k keys. Figure 9(a) plots the mean value, the 1st and the 99th percentiles of the path length, as a function of the network size. As expected, the mean path length increases logarithmically with the number of nodes. The same is also true for the 1st and the 99th percentiles. Figure 9(b) plots the PDF of the path length for a network with 2^{12} nodes. Remarkably, the maximum path length for this case has never exceeded 12 nodes in our simulation. In all the other cases we have obtained similar results. These results suggest that Chord's routing algorithm is fully scalable, and it will achieve good results in practice.

7.4 Simultaneous node failures

In this experiment, we consider the ability of the overlay network constructed by Chord to survive in the case of simultaneous node failures. This scenario can happen in practice when a LAN is temporary disconnected from the Internet, or a major network partition occurs. In particular, this experiment shows that the overlay network remains connected even when a large percentage of nodes fail simultaneously.

We consider again a 10^4 node network that stores 10^6 keys, and randomly select a percentage of p nodes that fail. Since there is no correlation between the node identifiers and the network topology, selecting a random number of nodes is equivalent to selecting all nodes from the same location or network partition. After the failures occurs, we wait for the network to reach steady state, and then measure the miss rate, i.e., the probability to successfully retrieve a key.

Figure 10(a) plots the mean miss rate and the 95% confidence interval as a function of the percentage of node failures, p. The miss rate increases linearly with p. Since this is exactly the miss rate due to the lost keys caused by node failures, we conclude that there is no significant partition in the overlay network. Indeed, if it were a half-to-half partition for example, we would expect that *half* of the requests to fail simply because in half of the cases the request initiator and the queried key



(a) (b)

Figure 10: (a) The key miss rate as a function of the percentage of node failures. (b) The key miss rate as a function of the rate at which nodes fails. This miss rate reflects *only* the query failures due to state inconsistency; it does *not* include query failures due to lost keys.

will be in different partitions. The fact that our results do not reflect this behavior, suggests that our algorithm is robust in the presence of simultaneous node failures.

7.5 Dynamic scenario

In a dynamic scenario, there are two reasons for which a query can fail. The first is because the node which stores the key has failed. The second is because nodes' finger tables and predecessors store inconsistent state due to concurrent joins, leaves and node failures. An interesting question is what is the impact of the concurrent operations and node failures on the miss rate. We try to answer this question with the next experiment.

To isolate between the two types of misses, when a node fails, we move its keys to its successor. In this way we factor out the misses due to lost keys. Any query failure in such a system will be then the result of table inconsistencies; nodes learn about the failed nodes only when they invoke the stabilization protocol. Also note that the simulator does not implement query retries. If a node forwards a query to a node and this node is down, the query simple fails. Thus, the results given in this section can be viewed as the worst case scenario for the query failures induced by state inconsistency.

Because the primary source of inconsistencies is node joinning and leaving, and because the main mechanism to resolve these inconsistencies in the described implementation is to invoke the stabilization protocol, Chord's performances will be sensitive to the frequency of node operations versus the frequency of invoking the stabilization protocol.

To illustrate this interdependence, we consider an experiment in which nodes join and fail randomly. During this time other nodes insert and search for random keys. Key insertions and lookups are generated according to a Poisson process at a rate of $1/5$ insertions per second, and one lookup per second, respectively. Similarly, joins and failures are modeled by a Poisson process with the mean arrival rate of R. We start with a network of 500 nodes storing 100 keys.

Figure 10(b) plots the average miss rates and the confidence intervals when the rate of node joining and leaving the network, R, is $0.01, 0.02, 0.05$, and 0.1, respectively. Note that 0.01 corresponds to one node joining and leaving every 100 seconds on average, while 0.1 corresponds to one node joining and leaving each second. For comparison, recall that each node invokes the stabilization protocol once every 30 sec on the average. The results presented in Figure 10(b) are averaged over approximately two hours of simulated time. The confidence intervals are computed over 10 independent runs. There are two points worth noting. First, as expected, the miss rate due to state inconsistency is much lower than the miss rate due to node failures (compare Figures 10(a) and (b), and consider the fact that during each simulation at least 14% of nodes fail on average). Second, the miss rate due to state inconsistency increases fast with failure frequency. This fully justifies the optimization described in Section 8 to reduce the time after which nodes hear about the node failures.

8 Chord system implementation

The Chord protocol simulated and analyzed in the previous sections has been implemented in an operational system.

8.1 Location table

Instead of limiting Chord's routing to just the information in the finger table, the Chord system also maintains a *location table*, which contains nodes that Chord has discovered recently while running the protocol. The location table is a cache that maps node identifiers to their *locations* (IP address and port). Node identifiers that are in the Chord finger table are pinned in the location table. Other nodes are replaced based on their network proximity. When replacing a node, the Chord server replaces a node that is far away in the network over a node that is close by in the network.

The location table is used to optimize lookup performance. Instead of choosing the node from the finger table that is the closest predecessor of the key (which might on the other side of the network), the Chord server chooses the node from the location table that is a close predecessor *and* that is close in the network (as measured by the round-trip time). Because of the location table's cache-replacement policy, which replaces far-away nodes over close-by nodes, a Chord server will learn over time about more and more nodes that are close by in the network and will use those nodes to resolve lookup queries.

When the Chord server learns about a new node, it inserts it in the location table. A Chord server learns about the location of a nodes as part of running the Chord protocol. A node identifier in a protocol message comes always along with its location. In addition to the location, the server records the source from which it learned about the new node. This server alerts the source when the server discovers that the node has failed.

The Chord server also records for each node in the location table the measured average round-trip time. Each time the server performs an RPC to a node, it measures the response time of the RPC and updates the average round-trip time to that node. Since all Chord RPCs are simple operations with small argument and result types (e.g., they don't recursively initiate new RPCs on the remote node), the round-trip time is dominated by network latency. (If the remote server happens to be overloaded because one particular key is popular, then we want to avoid the node anyway; either way the end-to-end measurement is helpful.)

Because the current implementation uses an iterative lookup procedure, a lookup request might still travel large distances over the network. Consider a server in Australia resolving a query that is destined to a server in the USA. The query might travel for a while close to Australia but once it makes the hop to the USA, it might take multiple hops back and forth between the Australia and the USA, because in our current implementation the node in Australia initiates all RPCs. We are considering switching from an iterative lookup to a recursive lookup procedure so that queries always travel in the direction of the their final destination. In that case, the protocol would return all the nodes that were visited to resolve a query to allow the initiator to build up a location table.

The location table is also used to recover quickly from failed nodes; as our simulation results have shown, this is an important optimization. When an RPC fails, the lookup procedure choses another node from the location table that is a close predecessor and routes queries through that node. Since over time, the server is likely to learn about quite a number of nodes, it is likely that it might be able to hop over failed nodes. When an RPC fails because of a node failure, the Chord server also deletes the node from its location table, and, if the node is also in its finger table, the server rebuilds the finger table.

To allow other servers also to learn quickly about failed nodes, the server alerts the node from which it learned about the failed node. A node that receives an alert RPC first checks whether it also observes that the node is down. (There might

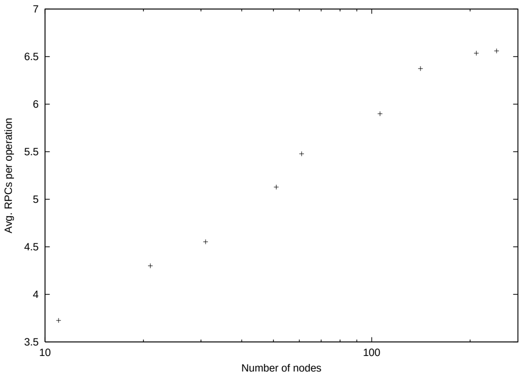


Figure 11: Average number of RPCs for a lookup in network scaling from 10 to 250 nodes.

be a network a problem that makes it impossible for a node A to talk to B, but node C might still be able to reach node B.) If the receiver of the alert message cannot contact the failed node either, it deletes the failed node from its location table, rebuilds its finger table (if necessary), and recursively alerts its sources.

8.2 Details

The Chord system consists of two programs: the client and and the server. The client program is a library that provides two key functions to the file sharing application: (1) it inserts values under a key and (2) it looks up values for a given key. It essentially implements the interface described in Section 3. To implement the inserts and lookups, the client calls its local Chord server.

The Chord server implements two interfaces: one to accept request from a local client and to communicate with other servers. Both interfaces are implemented as remote procedure calls. The exact message formats are described in the XDR protocol description language [20].

The file-sharing application uses the Chord system to store the mappings from file names to IP addresses of servers that store the file. The Chord system maps the file names into key identifiers with a cryptographic hash function (SHA-1). The value is an array of bytes, containing a list of IP addresses.

The Chord server internally represents key identifiers as multiple-precision integers to allow for keys that are larger than 64 bits. We use the GNU Multiple Precision Arithmetic Library to compute with large keys. Node identifiers are also represented as multiple-precision integers. The table that stores key-value pairs is implemented as a simple hash table, indexed by key.

The client and the server are user-level programs written in C++. The programs communicate with Sun RPC over a TCP connection. The Chord server sets up a TCP connection once with a remote server and sends many RPC over that connection. To handle many connections and RPCs simultaneously, the programs use SFS's asynchronous RPC library [15].

8.3 Experiment results

This implementation provides Chord with high-performance for its operations. For example, on a PIII 733, the Chord server can process 10,300 lookup RPCs per second.

We haven't deployed our servers in enough locations across the Internet yet to be able to collect meaningful data from a field experiment [3]. Instead, we validate the simulation results with the operational Chord service. Figure 11 shows the

[3] By the final version of this paper we hope to have results from a small-scale Internet deployment that confirms our proximity claims

number of RPCs per lookup with varying number of Chord nodes. As one can see the path lengths scale in the same manner as in our simulation results.

9 Conclusion

Many distributed applications need to determine the node that stores a data item. The Chord protocol solves this challenging problem in decentralized manner. It offers a powerful primitive: given a key, it will determine the node responsible for storing the key's value. The Chord protocol provides this primitive in an efficient way: in the steady state, in an N-node network, each node maintains routing information for only about $O(\log N)$ other nodes, and resolves all lookups via $O(\log N)$ messages to other nodes. Updates to the routing information for nodes leaving and joining require only $O(\log^2 N)$ messages.

We also present extensions to the Chord protocol that make it practical in actual systems. These extensions include support for concurrent joins, nodes leaving voluntarily and involuntarily, a high degree of fault tolerance, and minimizing the network distance that a query travels.

The Chord protocol and its extensions have been implemented in the Chord system. The Chord system uses the Chord primitive to provide a peer-to-peer lookup service that allows applications to insert and update key/value pairs and lookup values for a given key. The implementation is efficient (a single Chord node can process over 10,000 lookups per second) and scales well with the number of nodes (a lookup in a network of 250 nodes travels on average 7 hops).

Based on our experience using the Chord system for a peer-to-peer file sharing application and our results from theoretical analysis, simulation studies with up to 10,000 nodes, and experiments, we believe that the Chord protocol and system is a valuable component for many decentralized, large-scale distributed applications.

References

[1] ADJIE-WINOTO, W., SCHWARTZ, E. AND BALAKRISHNAN, H. AND LILLEY, J. The design and implementation of an intentional naming system. In *Proc. ACM Symposium on Operating Systems Principles* (Kiawah Island, SC, Dec. 1999), pp. 186–201.

[2] AUTHORS ELIDED FOR ANONYMITY. Building peer-to-peer systems with chord, a distributed location service. In *Submitted to 8th HotOS* (June 2001). This position paper is available on request.

[3] BAKKER, A., AMADE, E., BALLINTIJN, G., KUZ, I., VERKAIK, P., VAN DER WIJK, I., VAN STEEN, M., AND TANENBAUM., A. The globe distribution network. In *Proc. 2000 USENIX Annual Conf. (FREENIX Track)* (San Diego, June 2000), pp. 141–152.

[4] CHEN, Y., EDLER, J., GOLDBERG, A., GOTTLIEB, A., SOBTI, S., AND YIANILOS, P. A prototype implementation of archival intermemory. In *Proceedings of the fourth ACM Conference on Digital libraries (DL '99)* (1999).

[5] CLARKE, I. A distributed decentralised information storage and retrieval system. Master's thesis, University of Edinburgh, 1999.

[6] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability* (Berkeley, California, June 2000). http://freenet.sourceforge.net.

[7] C.PLAXTON, RAJARAMAN, R., AND RICHA, A. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA* (Newport, Rhode Island, June 1997), pp. 311–320.

[8] FLOYD, S., AND JACOBSON, V. The synchronization of periodic routing messages. In *Proceedings of ACM SIGCOMM'93* (San Francisco, CA, Sept. 1993), pp. 33–44.

[9] Gnutella website. http://gnutella.wego.com.

[10] Jini (TM). http://java.sun.com/products/jini/, 2000.

[11] KARGER, D., LEHMAN, E., LEIGHTON, F., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (May 1997), pp. 654–663.

[12] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO., B. Oceanstore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)* (Boston, MA, November 2000).

[13] LEWIN, D. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, MIT, 1998. Available at the MIT Library, `http://thesis.mit.edu`.

[14] LI, J., JANNOTTI, J., COUTO, D. S. J. D., KARGER, D. R., AND MORRIS, R. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (MobiCom '00)* (Boston, Massachusetts, August 2000), pp. 120–130.

[15] MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)* (Kiawah Island, South Carolina, December 1999). `http://www.fs.net`.

[16] MOCKAPETRIS, P., AND DUNLAP, K. J. Development of the Domain Name System. In *Proc. ACM SIGCOMM* (Stanford, CA, 1988).

[17] MOTWANI, R., AND RAGHAVAN, P. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995.

[18] Napster. http://www.napster.com.

[19] Ohaha. http://www.ohaha.com/design.html.

[20] SRINIVASAN, R. XDR: External data representation standard. RFC 1832, Network Working Group, August 1995.

[21] Universal Plug and Play: Background. http://www.upnp.com/resources/UPnPbkgnd.htm, 2000.

[22] VAN STEEN, M., HAUCK, F., BALLINTIJN, G., AND TANENBAUM, A. Algorithmic design of the globe wide-area location service. *The Computer Journal 41*, 5 (1998), 297–310.

[23] VEIZADES, J., GUTTMAN, E., PERKINS, C., AND KAPLAN, S. *Service Location Protocol*, June 1997. RFC 2165 (http://www.ietf.org/rfc/rfc2165.txt).