ISSN 2277-2685
IJESR/Feb. 2021/ Vol-11/Issue-1/1-10
Boddapati Soujanya *et. al.*, / International Journal of Engineering & Science Research

Scalable P2P Resource Locator (SPRL)

[1] Boddapati Soujanya, [2] Dr.T.Muni Sankar [3] Yezarla Ashok

[1] Associate Professor, Department of CSE, Rise Krishna Sai Gandhi Group of Institutions,[2] Associate Professor, Department of CSE, Rise Krishna Sai Gandhi Group of Institutions, [3] Assistant Professor, Department of CSE, Rise Krishna Sai Gandhi Group of Institutions

Abstract

The Scalable P2P Resource Locator (SPRL) is a fundamental component in peer-to-peer (P2P) network architectures designed to efficiently and effectively locate resources across distributed networks. This abstract provides an overview of SPRL's key objectives, design principles, and functionalities.In the realm of Internet applications and distributed systems, the need for efficient resource discovery in a scalable and decentralized manner is paramount. SPRL addresses this challenge by offering a sophisticated framework for locating and accessing resources across a dynamic network of peers. Its design is underpinned by the following core principles: Scalability: SPRL is architected to seamlessly accommodate an ever-expanding network of peers and resources. It leverages techniques such as distributed hash tables (DHTs) and peer routing algorithms to ensure that the system can handle growth without compromising performance.

This abstract serves as an introduction to the Scalable P2P Resource Locator (SPRL) framework, which plays a vital role in the modern landscape of internet applications. By adhering to these principles, SPRL empowers distributed systems to efficiently locate and access resources while maintaining scalability, decentralization, efficiency, and fault tolerance.

1. Introduction

In the dynamic landscape of the internet and distributed systems, the efficient and decentralized discovery of resources is a critical necessity. Peer-to-peer (P2P) networks have emerged as a versatile solution for various applications, from file sharing to content distribution, and their effectiveness depends heavily on the ability to locate and access resources seamlessly. The Scalable P2P Resource Locator (SPRL) addresses this core challenge with ingenuity and sophistication.At its core, SPRL represents a fundamental building block in P2P network architectures, focusing on the seamless and efficient location of resources across a vast and ever-evolving network of peers. This introduction provides an overview of SPRL's objectives, significance, and the principles that underpin its design.

As the digital world continues to grow, the volume of data and resources available across the internet has expanded exponentially. Whether it's for retrieving files, searching for specific content, or accessing distributed services, the ability to locate resources swiftly and efficiently is a cornerstone of internet applications.

Traditional client-server models, while effective, have limitations when it comes to scalability, fault tolerance, and centralization. These constraints have given rise to P2P networks, where each participant, or peer, collaboratively contributes to the network's operation. However, P2P systems must grapple with the challenge of enabling peers to discover and access resources distributed across the network, a task that SPRL excels at.

SPRL operates in alignment with several core principles:

Scalability: The system is designed to scale seamlessly as more peers and resources join the network. This scalability is achieved through the use of distributed hash tables (DHTs) and intelligent peer routing algorithms that enable the system to grow without sacrificing performance.

Decentralization: SPRL operates in a fully decentralized manner. It eliminates the need for centralized servers, ensuring that resource location requests are resolved directly between peers. This decentralization enhances the robustness and availability of the system.

Efficiency: SPRL prioritizes efficiency in resource discovery. It employs optimization strategies such as caching and proximity-based routing to minimize query and retrieval latency, ensuring that users can quickly access the resources they seek.Fault Tolerance: The system is built to withstand node failures, network disruptions, and other potential issues that can arise in a P2P environment. Redundancy, replication, and self-healing mechanisms are deployed to maintain system integrity.

2. Related Work

While Chord maps keys onto nodes, traditional name and lo- cation services provide a *direct* mapping between keys and val- ues. A value can be an address, a document, or an arbitrary data item. Chord can easily implement this functionality by storing each key/value pair at the node to which that key maps. For this reason and to make the comparison clearer, the rest of this section assumesa Chord-based service that maps keys onto values.
DNS provides a host name to IP address mapping [15]. Chord can provide the same service with the name representing the key and the associated IP address representing the value. Chord re- quires no special servers, while DNS relies on a set of special root

servers. DNS names are structured to reflect administrative bound-aries; Chord imposes no naming structure. DNS is specialized tothe task of finding named hosts or services, while Chord can alsobe used to find data objects that are not tied to particular machines.The Freenet peer-to-peer storage system [4, 5], like Chord, is decentralized and symmetric and automatically adapts when hostsleave and join. Freenet does not assign responsibility for docu- ments to specific servers; instead, its lookups take the form ofsearches for cached copies. This allows Freenet to provide a degreeof anonymity, but prevents it from guaranteeing retrieval of existingdocuments or from providing low bounds on retrieval costs. Chorddoes not provide anonymity, but its lookup operation runs in pre- dictable time and always results in success or definitive failure.

The Ohaha system uses a consistent hashing-like algorithm for mapping documents to nodes, and Freenet-style query routing [18].As a result, it shares some of the weaknesses of Freenet. Archival Intermemory uses an off-line computed tree to map logical ad- dresses to machines that store the data [3].
The Globe system [2] has a wide-area location service to map ob- ject identifiers to the locations of moving objects. Globe arranges the Internet as a hierarchy of geographical, topological, or adminis- trative domains, effectively constructing a static world-wide searchtree, much like DNS. Information about an object is stored in a particular leaf domain, and pointer caches provide search short cuts [22]. The Globe system handles high load on the logical root by partitioning objects among multiple physical root servers us- ing hash-like techniques. Chord performs this hash function well enough that it can achieve scalability without also involving any hierarchy, though Chord does not exploit network locality as well as Globe.

The distributed data location protocol developed by Plaxton *et al.* [19], 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 makea logarithmic number hops and that keys are well balanced, but thePlaxton protocol also ensures, subject to assumptions about net- work topology, that queries never travel further in network distance than the node where the key is stored. The advantage of Chord is that it is substantially less complicated and handles concurrent node joins and failures well. The Chord protocol is also similar to Pastry, the location algorithm used in PAST [8]. However, Pastry is a prefix-based routing protocol, and differs in other details from Chord.

CAN uses a -dimensional Cartesian coordinate space (for somefixed) to implement a distributed hash table that maps keys ontovalues [20]. Each node maintains state, and the lookup costis . Thus, in contrast to Chord, the state maintained by aCAN node does not depend on the network size , but the lookup cost increases faster than . I, CAN lookup timesand storage needs match Chord's. However, CAN is not designed to vary as (and thus) varies, so this match will only occur for the "right" corresponding to the fixed . CAN requires an additional maintenance protocol to periodically remap the identifier space onto nodes. Chord also has the advantage that its correctnessis robust in the face of partially incorrect routing information.
Chord's routing procedure may be thought of as a one- dimensional analogue of the Grid location system [14]. Grid relieson real-world geographic location information to route its queries;Chord maps its nodes to an artificial

one-dimensional space withinwhich routing is carried out by an algorithm similar to Grid's.

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

possess [17], and the lack of scalability that systems like Gnutella display because of their widespread use of broadcasts [10].

3. System Model

Chord simplifies the design of peer-to-peer systems and applica- tions based on it by addressing these difficult problems:

Load balance: Chord acts as a distributed hash function, spreading keys evenly over the nodes; this provides a degreeof natural load balance.
Decentralization: Chord is fully distributed: no node is more important than any other. This improves robustness and makes Chord appropriate for loosely-organized peer-to-peerapplications.
Scalability: The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible.No parameter tuning is required to achieve this scaling.
Availability: Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensur-ing that, barring major failures in the underlying network, the node responsible for a key can always be found. This is trueeven if the system is in a continuous state of change.

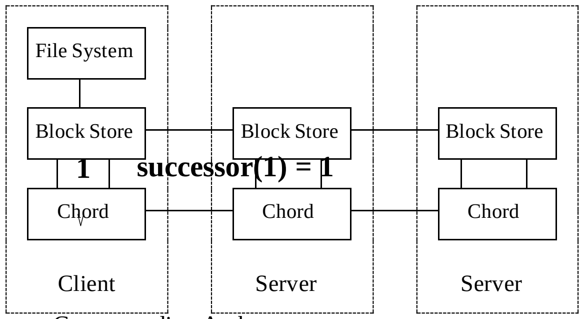
Flexible naming: Chord places no constraints on the struc- ture of the keys it looks up: the Chord key-space is flat. Thisgives applications a large amount of flexibility in how they map their own names to Chord keys.
The Chord software takes the form of a library to be linked with the client and server applications that use it. The application in- teracts with Chord in two main ways. First, Chord provides a lookup(key) algorithm that yields the IP address of the node responsible for the key. Second, the Chord software on each node notifies the application of changes in the set of keys that the node is responsible for. This allows the application software to, for ex- ample, move corresponding values to their new homes when a newnode joins.
The application using Chord is responsible for providing any de-sired authentication, caching, replication, and user-friendly naming of data. Chord's flat key space eases the implementation of these features. For example, an application could authenticate data by storing it under a Chord key derived from a cryptographic hash of the data. Similarly, an application could replicate data by storing it under two distinct Chord keys derived from the data's application-level identifier.

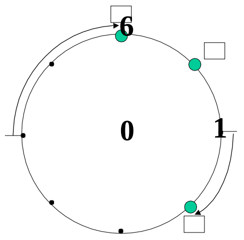
The following are examples of applications for which Chord would provide a good foundation:

Cooperative Mirroring, as outlined in a recent proposal [6].Imagine a set of software developers, each of whom wishes to publish a distribution. Demand for each distribution mightvary dramatically, from very popular just after a new release to relatively unpopular between releases. An efficient ap- proach for this would be for the developers to cooperatively mirror each others' distributions. Ideally, the mirroring sys- tem would balance the load across all servers, replicate and cache the data, and ensure authenticity. Such a system should be fully decentralized in the interests of reliability, and be- cause there is no natural central administration.

Time-Shared Storage for nodes with intermittent connectivity. If a person wishes some data to be always available, but their

7 1 **successor(1) = 1**





6
successor(6) = 0

v ^ v ^ v ^

5> < >

Figure 1: Structure of an example Chord-based distributedstorage system.

machine is only occasionally available, they can offer to store others' data while they are up, in return for having their data stored elsewhere when they are down. The data's name can serve as a key to identify the (live) Chord node responsible for storing the data item at any given time. Many of the same issues arise as in the Cooperative Mirroring applica- tion, though the focus here is on availability rather than loadbalance.

Distributed Indexes to support Gnutella- or Napster-like keyword search. A key in this application could be derived from the desired keywords, while values could be lists of machines offering documents with those keywords.

Figure 2: An identifier circle consisting of the three nodes 0, 1, and 3. In this example, key 1 is located at node 1, key 2 at node3, and key 6 at node 0.

Chord improves the scalability of consistent hashing by avoid- ing the requirement that every node know about every other node. A Chord node needs only a small amount of "routing" informa- tion about other nodes. Because this information is distributed, a node resolves the hash function by communicating with a few othernodes. In an - node network, each node maintains information only about other nodes, and a lookup requires messages.

Chord must update the routing information when a node joins or leaves the network; a join or leave requires messages.

4. The Base Chord Protocol

The Chord protocol specifies how to find the locations of keys, how new nodes join the system, and how to recover from the failure(or planned departure) of existing nodes. This section describes a simplified version of the protocol that does not handle concurrent joins or failures. Section 5 describes enhancements to the base pro-tocol to handle concurrent joins and failures.

Overview

At its heart, Chord provides fast distributed computation of ahash function mapping keys to nodes responsible for them. It uses*consistent hashing* [11, 13], which has several good properties.With high probability the hash function balances load (all nodesreceive roughly the same number of keys). Also with high prob-ability, when anode joins (or leaves) the network, only anfraction of the keys are moved to a different location—
this is clearly the minimum necessary to maintain a balanced load.
identifier is produced by hashing the key. We will use the term "key" to refer to both the original key and its image under the hashfunction, as the meaning will be clear from context. Similarly, theterm "node" will refer to both the node and its identifier under thehash function. The identifier length must be large enough to make the probability of two nodes or keys hashing to the same iden- tifier negligible.

Consistent hashing assigns keys to nodes as follows. Identifiersare ordered in an *identifier circle* modulo . Key is assigned to the first node whose identifier is equal to or follows (the identifierof) in the identifier space. This node is called the *successor node*of key , denoted by *successor* If identifiers are represented asa circle of numbers from to, thenis thefirst node clockwise from .

Figure 2 shows an identifier circle with. The circle hasthree nodes: 0, 1, and 3. The successor of identifier 1 is node 1, sokey 1 would be located at node 1. Similarly, key 2 would be locatedat node 3, and key 6 at node 0.

Consistent hashing is designed to let nodes enter and leave the network with minimal disruption. To maintain the consistent hash-ing mapping when a node joins the network, certain keys previ- ously assigned to 's successor now become assigned to . When node leaves the network, all of its assigned keys are reassigned to 's successor.

No other changes in assignment of keys to nodesneed occur. In the example above, if a node were to join with iden-tifier 7, it would capture the key with identifier 6 from the node with identifier 0.

The following results are proven in the papers that introduced consistent hashing [11, 13]:

THEOREM 1. *For any set of nodes and keys, with high probability:*

1. *Each node is responsible for at most keys*
2. *When annode joins or leaves the network, respon-sibility forkeys changes hands (and only to or fromthe joining or leaving node).*

When consistent hashing is implemented as described above, the theorem proves a bound of. The consistent hashing paper shows that can be reduced to an arbitrarily small constant by having each node run"virtual nodes" each with itsown identifier.

The phrase "with high probability" bears some discussion. A simple interpretation is that the nodes and keys are randomly cho-sen, which is plausible in a non-adversarial model of the world. The probability distribution is then over random choices of keys and nodes, and says that such a random choice is unlikely to pro-

Notation	Definition
finger	mod ,
interval	*finger start finger start*
	first node *finger start*
successor	the next node on the identifier circle; *finger node*
predecessor	the previous node on the identifier circle

Table 1: Definition of variables for node , using -bit identi-fiers.

duce an unbalanced distribution. One might worry, however, aboutan adversary who intentionally chooses keys to all hash to the same identifier, destroying the load balancing property. The consistent hashing paper uses " -universal hash functions" to provide certainguarantees even in the case of nonrandom keys.

Rather than using a -universal hash function, we chose to usethe standard SHA-1 function as our base hash function. This makes our protocol deterministic, so that the claims of "high probability" no longer make sense. However, producing a set of keys that collide under SHA-1 can be seen, in some sense, as inverting, or "decrypt-ing" the SHA-1 function. This is believed to be hard to do. Thus,instead of stating that our theorems hold with high probability, wecan claim that they hold "based on standard hardness assumptions."For simplicity (primarily of presentation), we dispense with the use of virtual nodes. In this case, the load on a node may exceed theaverage by (at most) anfactor with high probability (orin our case, based on standard hardness assumptions). One reasonto avoid virtual nodes is that the number needed is determined bythe number of nodes in the system, which may be difficult to deter-mine. Of course, one may choose to use an a priori upper bound onthe number of nodes in the system; for example, we could postulateat most one Chord server per IPv4 address. In this case running 32 virtual nodes per physical node would provide good load balance.

In the section reporting our experimental results (Section 6), we will observe (and justify) that the average lookup time is $^{-}$.

Node Joins

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

1. Each node's successor is correctly maintained.
2. For every key , node is responsible for .

In order for lookups to be fast, it is also desirable for the finger tables to be correct.

This section shows how to maintain these invariants when a sin-gle node joins. We defer the discussion of multiple nodes joining simultaneously to Section 5, which also discusses how to handle a node failure. Before describing the join operation, we summa- rize its performance (the proof of this theorem is in the companion technical report [21]):

THEOREM 3. *With high probability, any node joining or leav- ing an -node Chord network will use messages to re-establish the Chord routing invariants and finger tables.*

To simplify the join and leave mechanisms, each node in Chord maintains a *predecessor pointer*. A node's predecessor pointer con-tains the Chord identifier and IP address of the immediate predeces- sor of that node, and can be used to walk counterclockwise around the identifier circle.

To preserve the invariants stated above, Chord must perform three tasks when a node joins the network:

1. Initialize the predecessor and fingers of node .
2. Update the fingers and predecessors of existing nodes to re-flect the addition of .
3. Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node is now respon-sible for.

We assume that the new node learns the identity of an existing Chord node by some external mechanism. Node uses to

5. Concurrent Operations and Failures

In practice Chord 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 algorithms de- scribed in Section 4 to handle these situations.

Stabilization

The join algorithm in Section 4 aggressively maintains the finger tables of all nodes as the network evolves. Since this invariant is difficult to maintain in the face of concurrent joins in a large net- work, we separate our correctness and performance goals. A basic "stabilization" protocol is used to keep nodes' successor pointers up to date, which is sufficient to guarantee correctness of lookups. Those successor pointers are then used to verify and correct fin- ger table entries, which allows these lookups to be fast as well as correct.

If joining nodes have affected some region of the Chord ring, a lookup that occurs before stabilization has finished can exhibit one of three behaviors. The common case is that all the finger ta- ble entries involved in the lookup are reasonably current, and the lookup finds the correct successor in steps. The second case is where successor pointers are correct, but fingers are inaccu-rate. This yields correct lookups, but they may be slower. In the final case, the nodes in the affected region have incorrect successor pointers, or keys may not yet have migrated to newly joined nodes, and the lookup may fail. The higher-layer software using Chord will notice that the desired data was not found, and has the option of retrying the lookup after a pause. This pause can be short, since stabilization fixes successor pointers quickly.

Our stabilization scheme guarantees to add nodes to a Chord ring in a way that preserves reachability of existing nodes, even in the face of concurrent joins and lost and reordered messages. Stabi- lization by itself won't correct a Chord system that has split into multiple disjoint cycles, or a single cycle that loops multiple times around the identifier space. These pathological cases cannot be produced by any sequence of ordinary node joins. It is unclear whether they can be produced by network partitions and recoveries or intermittent failures. If produced, these cases could be detected and repaired by periodic sampling of the ring topology.

Figure 7 shows the pseudo-code for joins and stabilization; this code replaces Figure 6 to handle concurrent joins. When node first starts, it calls *join*, where is any known Chord node. The function asks to find the immediate successor of . By itself, does not make the rest of the network aware of .

Every node runs *stabilize* periodically (this is how newly joined nodes are noticed by the network). When node runs *stabilize*, it asks 's successor for the successor's predecessor , and de- cides whether should be 's successor instead. This would be the case if node recently joined the system. *stabilize* also noti- fies node 's successor of 's existence, giving the successor the chance to change its predecessor to . The successor does this onlyif it knows of no closer predecessor than .

As a simple example, suppose node joins the system, and itsID lies between nodes and would acquire as its succes-sor. Node , when notified by , would acquire as its predeces- sor. When next runs *stabilize*, it will ask for its predecessor(which is now); would then acquire as its successor. Finally,will notify , and will acquire as its predecessor. At this point, all predecessor and successor pointers are correct

As soon as the successor pointers are correct, calls to *find predecessor* (and thus *find successor*) will work. Newly joined nodes that have not yet been fingered may cause *find predecessor* to initially undershoot, but the loop in the lookup algorithm will nev-ertheless follow successor (*finger*) pointers through the newly joined nodes until the correct predecessor is reached. Eventually *fix fingers* will adjust finger table entries, eliminating the need for these linear scans.

The following theorems (proved in the technical report [21]) show that all problems caused by concurrent joins are transient. The theorems assume that any two nodes trying to communicate will eventually succeed.

THEOREM 4. *Once a node can successfully resolve a given query, it will always be able to do so in the future.*

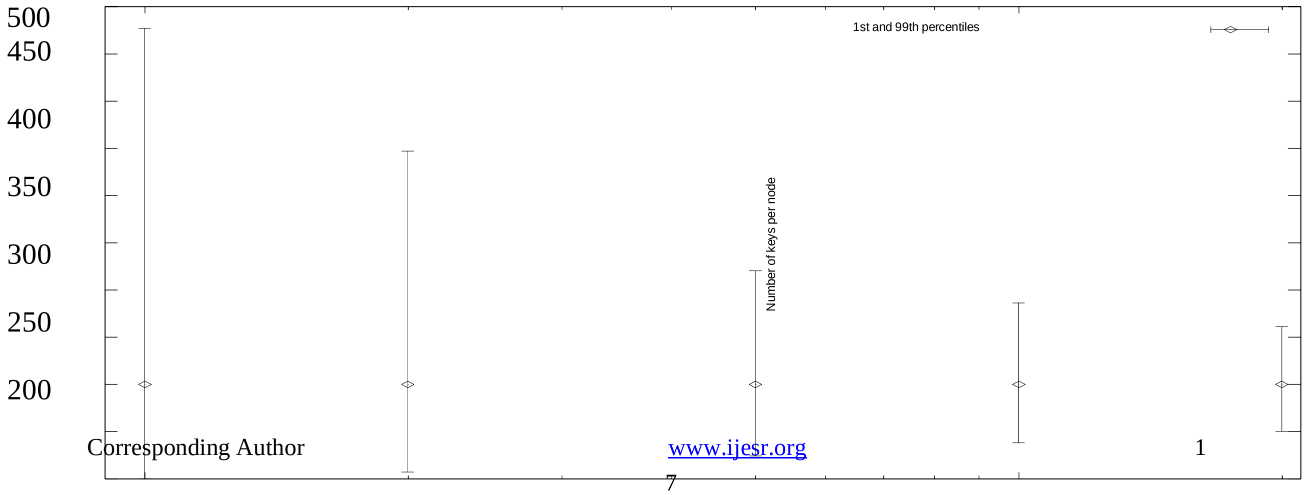
THEOREM 5. *At some time after the last join all successor pointers will be correct.*

The proofs of these theorems rely on an invariant and a termina- tion argument. The invariant states that once node can reach nodevia successor pointers, it always can. To argue termination, we consider the case where two nodes both think they have the same successor . In this case, each will attempt to notify , and will eventually choose the closer of the two (or some other, closer node)as its predecessor. At this point the farther of the two will, by con- tacting , learn of a better successor than . It follows that every node progresses towards a better and better successor over time. This progress must eventually halt in a state where every node is considered the successor of exactly one other node; this defines a cycle (or set of them, but the invariant ensures that there will be at
most one).

We have not discussed the adjustment of fingers when nodes join because it turns out that joins don't substantially damage the per- formance of fingers. If a node has a finger into each interval, then these fingers can still be used even after joins. The distance halving argument is essentially unchanged, showing thathopssuffice to reach a node "close" to a query's target. New joins in- fluence the lookup only by getting in between the old predecessor and successor of a target query. These new nodes may need to be scanned linearly (if their fingers are not yet accurate). But unless a

tremendous number of nodes joins the system, the number of nodes between two old nodes is likely to be very small, so the impact onlookup is negligible. Formally, we can state the following:

THEOREM 6. *If we take a stable network with nodes, and another set of up to nodes joins the network with no finger point-ers (but with correct successor pointers), then lookups will still take time with high probability.*



150
More generally, so long as the time it takes to adjust fingers is less than the time it takes the network to double in size, lookups should continue to take hops. Failures and Replication

When a nodefails, nodes whose finger tables include mustfind 's successor. In addition, the failure of must not be allowed to disrupt queries that are in progress as the system is re-stabilizing. The key step in failure recovery is maintaining correct succes- sor pointers, since in the worst case *find predecessor* can make progress using only successors. To help achieve this, each Chord node maintains a "successor-list" of its nearest successors on the Chord ring. In ordinary operation, a modified version of the *stabi-lize* routine in Figure 7 maintains the successor-list. If node no-tices that its successor has failed, it replaces it with the first live en-try in its successor list. At that point, can direct ordinary lookups for keys for which the failed node was the successor to the new successor. As time passes, *stabilize* will correct finger table entries
and successor-list entries pointing to the failed node.

After a node failure, but before stabilization has completed, other nodes may attempt to send requests through the failed node as part of a *find successor* lookup. Ideally the lookups would be able to proceed, after a timeout, by another path despite the failure. In many cases this is possible. 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, nodes in the successor-list are also available as alternates.

The technical report proves the following two theorems that show that the successor-list allows lookups to succeed, and be effi-cient, even during stabilization [21]:

THEOREM 7. *If we use a successor list of length*
in a network that is initially stable, and then every node fails with probability 1/2, then with high probability find successor returns the closest living successor to the query key.

THEOREM 8. *If we use a successor list of length*
in a network that is initially stable, and then every node fails with probability 1/2, then the expected time to execute find successor in the failed network is .

The intuition behind these proofs is straightforward: a node's successors all fail with probability , so with high prob-ability a node will be aware of, so able to forward messages to, its closest living successor.

The successor-list mechanism also helps higher layer software replicate data. A typical application using Chord might store repli-cas of the data associated with a key at the nodes succeeding the key. The fact that a Chord node keeps track of its successors means that it can inform the higher layer software when successors come and go, and thus when the software should propagate new replicas.

6. Simulation and Experimental Results

In this section, we evaluate the Chord protocol by simulation. The simulator uses the lookup algorithm in Figure 4 and a slightly older version of the stabilization algorithms described in Section 5. We also report on some preliminary experimental results from an operational Chord-based system running on Internet hosts.

Protocol Simulator

The Chord protocol can be implemented in an *iterative* or *recur-sive* style. In the iterative style, a node resolving a lookup initiates all communication: it asks a series of nodes for information from their finger tables, each time moving closer on the Chord ring to the desired successor. In the recursive style, each intermediate node forwards a request to the next node until it reaches the successor. The simulator implements the protocols in an iterative style.

Load Balance

We first consider the ability of consistent hashing to allocate keys to nodes evenly. In a network with nodes and keys we would like the distribution of keys to nodes to be tight around .

We consider a network consisting of nodes, and vary the total number of keys from to in increments of . For each value, we repeat the experiment 20 times. Figure 8(a) plots the mean and the 1st and 99th percentiles 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 some nodes store no keys. To clarify this, Figure 8(b) plots the probability density function (PDF) of the number of keys per node when there are keys stored in the network. The maximum number of nodes stored by any node in this case is 457, or the mean value. For comparison, the 99th percentile is the mean value.

One reason for these variations is that node identifiers do not uni- formly cover the entire identifier space. If we divide the identifier space in equal-sized bins, where is the number of nodes, then we might hope to see one node in each bin. But in fact, the proba- bility that a particular bin does not contain any node is. For large values of this approaches .

As we discussed earlier, the consistent hashing paper solves this problem by associating keys with virtual nodes, and mapping mul- tiple virtual nodes (with unrelated identifiers) to each real node. Intuitively, this will provide a more uniform coverage of the iden- tifier space. For example, if we allocate randomly chosen virtual nodes to each real node, with high probability each of the

7. Conclusion

Many distributed peer-to-peer applications need to determine the node that stores a data item. The Chord protocol solves this challenging problem in decentralized manner. It offers a power- ful primitive: given a key, it determines the node responsible for storing the key's value, and does so efficiently. In the steady state, in an-node network, each node maintains routing information for only about other nodes, and resolves all lookups via messages to other nodes. Updates to the routing infor- mation for nodes leaving and joining require only mes sages.

Attractive features of Chord include its simplicity, provable corectness, and provable performance even in the face of concurrent node arrivals and departures. It continues to function correctly, al- beit at degraded performance, when a node's information is only partially correct. Our theoretical analysis, simulations, and exper- imental results confirm that Chord scales well with the number of nodes, recovers from large numbers of simultaneous node failures and joins, and answers most lookups correctly even during recov- ery.

We believe that Chord will be a valuable component for peer- to-peer, large-scale distributed applications such as cooperative file sharing, time-shared available storage systems, distributed indices for document and service discovery, and large-scale distributed computing platforms.

Acknowledgments

We thank Frank Dabek for the measurements of the Chord proto- type described in Section 6.6, and David Andersen for setting up the testbed used in those measurements.

8. References

[1] ANDERSEN, D. Resilient overlay networks. Master's thesis, Department of EECS, MIT, May 2001. http://nms.lcs.mit.edu/projects/ron/.

[2] 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, CA, June 2000), pp. 141–152.

[3] CHEN, Y., EDLER, J., GOLDBERG, A., GOTTLIEB, A., SOBTI, S., AND YIANILOS, P. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital libraries* (Berkeley, CA, Aug. 1999), pp. 28–37.

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

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

[6] DABEK, F., BRUNSKILL, E., KAASHOEK, M. F., KARGER, D., MORRIS, R., STOICA, I., AND BALAKRISHNAN, H. Building peer-to-peer systems with Chord, a distributed location service. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Elmau/Oberbayern, Germany, May 2001), pp. 71–76.

[7] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative

[8] storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (To appear; Banff, Canada, Oct. 2001).

[9] DRUSCHEL, P., AND ROWSTRON, A. Past: Persistent and anonymous storage in a peer-to-peer networking environment. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS 2001)* (Elmau/Oberbayern, Germany, May 2001), pp. 65–70.

[10] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Apr. 1995.

[11] Gnutella. http://gnutella.wego.com/.

[12] 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* (El Paso, TX, May 1997), pp. 654–663.