Full-Information Lookups for Peer-to-Peer Overlays

Pedro Fonseca, Rodrigo Rodrigues, Anjali Gupta, and Barbara Liskov, Member, IEEE

Abstract—Most peer-to-peer lookup schemes keep a small amount of routing state per node, typically logarithmic in the number of overlay nodes. This design assumes that routing information at each member node must be kept small so that the bookkeeping required to respond to system membership changes is also small, given that aggressive membership dynamics are expected. As a consequence, lookups have high latency as each lookup requires contacting several nodes in sequence. In this paper, we question these assumptions by presenting a peer-to-peer routing algorithm with small lookup paths. Our algorithm, called “OneHop,” maintains full information about the system membership at each node, routing in a single hop whenever that information is up to date and in a small number of hops otherwise. We show how to disseminate information about membership changes quickly enough so that nodes maintain accurate complete membership information. We also present analytic bandwidth requirements for our scheme that demonstrate that it could be deployed in systems with hundreds of thousands of nodes and high churn. We validate our analytic model using a simulated environment and a real implementation. Our results confirm that OneHop is able to achieve high efficiency, usually reaching the correct node directly 99 percent of the time.

Index Terms—Distributed systems, peer to peer, network topology, routing protocol, distributed data structures.

1 INTRODUCTION

Structured peer-to-peer overlays like Chord [2], CAN [3], Pastry [4], or Tapestry [5] provide a substrate for building large-scale distributed applications. These overlays allow applications to locate objects stored in the system in a limited number of overlay hops.

Peer-to-peer lookup algorithms strive to maintain a small amount of per-node routing state—typically $O(\log N)$—because their designers expect that system membership changes frequently. This expectation has been confirmed for successfully deployed systems. A 2002 study of Gnutella [6] shows that the average session time was only 2.9 hours. This is equivalent to saying that in a system with 100,000 nodes, there are about 19 membership change events per second. More recent studies confirm that session times continue to be very short [7], [8].

Maintaining small tables helps keep the amount of bookkeeping required to deal with membership changes small. However, there is a price to pay: lookups have high latency since each lookup requires contacting several nodes in sequence.

This paper questions the need to keep routing state small. We take the position that maintaining full routing state (i.e., a complete description of the system membership at each node) is viable even in a very large system, e.g., containing hundreds of thousands of nodes. We present techniques that show that in systems of this size, nodes can maintain membership information accurately, yet the communication costs are low. The results imply that a peer-to-peer system can route very efficiently even though the system is large and membership is changing rapidly.

We present a peer-to-peer lookup system called OneHop that maintains complete membership information at each node. We show analytic results that prove that the system meets our goals of reasonable accuracy and bandwidth usage. It is, of course, easy to achieve these goals for small systems. Our algorithm is designed to scale to large systems. Our analysis shows that we are able to use OneHop for systems of up to hundreds of thousands of nodes.

The paper presents simulation results that validate what our analytic models predict. These results show that performance does not degrade significantly as the system becomes larger nor in the presence of high churn.

We implemented the OneHop protocol as an additional protocol supported by the Chord framework for developing peer-to-peer applications [9]. We evaluated our implementation on a network that emulated wide-area internode latencies (using ModelNet [10]), and our results corroborate the evaluation performed using simulations.

1. This submission is based on a paper previously published in the First Symposium on Networked Systems Design and Implementation (NSDI) [1]. It extends the previous publication by presenting a complete implementation of the protocol and an evaluation of the protocol using that implementation. In the previous paper, we only had simulated the protocol. Other improvements include a revision of the analysis in Section 4 and an overall text revision.


Recommended for acceptance by X. Zhang.

For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2007-11-0429. Digital Object Identifier no. 10.1109/TPDS.2008.222.
The rest of the paper is organized as follows: Section 2 presents our system model. Section 3 describes the OneHop protocol. Section 4 shows how to parameterize the system and presents an analysis of the bandwidth requirements of this scheme. Section 5 presents an evaluation of OneHop using simulations. Section 6 presents the implementation of the protocol and an evaluation using that implementation. Section 7 discusses related work. We conclude with a discussion of what we have accomplished.

2 System Model

We consider a system of \( n \) nodes, where \( n \) is a large number like \( 10^5 \) or \( 10^6 \). We assume dynamic membership behavior as in Gnutella, which is representative of an open Internet environment. Participants of this system have an average session time of approximately 2.9 hours [6].

From the average session times in Gnutella, we deduce that systems of \( 10^5 \) and \( 10^6 \) nodes would show around 20 and 200 membership changes per second, respectively. We call this rate \( r \). We refer to membership changes as events in the rest of the paper.

Every node in the overlay is assigned a random node identifier in a large identifier space (in our case, we use 160-bit node IDs). Identifiers are ordered in an identifier ring modulo \( 2^{160} \). We assume that identifiers are generated such that the resulting set is uniformly distributed in the identifier space, for example, by setting a node’s identifier to be the cryptographic hash of its network address. Every node has a predecessor and a successor in the identifier ring, and it periodically sends keep-alive messages to these nodes.

Similarly, each item has a key, which is also an identifier in the ring. Responsibility for an item (e.g., providing storage for it) rests with its successor; this is the first node in the identifier ring clockwise from key. This mapping from keys to nodes is the same as the one used in Chord [2], but changing our system to use other mappings is straightforward.

Clients issue queries that try to reach the successor node of a particular identifier. We intend our system to satisfy a large fraction, \( f \), of the queries correctly on the first attempt (where each attempt requires one hop). Our goal is to support high values of \( f \), e.g., \( f = 0.99 \). A query may fail in its first attempt due to a membership change if the notification of the change has not reached the querying node. In such a case, the query can still be rerouted and succeed in a larger number of hops. Nevertheless, we define failed queries as those that are not answered correctly in the first attempt, as our objective is to have one-hop lookups.

3 OneHop Design

This section presents the design of OneHop. In this scheme, every node maintains a full routing table containing information about every other node in the overlay.

Section 3.1 describes how the algorithm handles membership changes, namely, how to convey information about these changes to all the nodes in the ring. Section 3.2 explains how the algorithm reacts to node failures and presents an informal correctness argument for our approach. Section 3.3 discusses issues about asymmetry in the load of individual nodes. Section 3.4 explains how the algorithm deals with a sparsely populated network.

3.1 Membership Changes

Membership changes (caused by nodes starting or ending a session) raise two important issues that our algorithm must address. First, we must update local information about the membership change, in order for each node in the system to determine precisely which interval in the ID space it is responsible for. The second issue is conveying information about the change to all the nodes in the ring so that these nodes will maintain correct information about the system membership and consequently manage to route in a single hop.

To maintain correct local information (i.e., information about each node’s successor and predecessor node), we use the algorithm developed for maintaining a correct successor and predecessor pointer in Chord [2]: every node \( n \) runs a stabilization routine periodically, wherein it sends keep-alive messages to its successor \( s \) and predecessor \( p \). Node \( s \) checks if \( n \) is indeed its predecessor, and if not, it notifies \( n \) of the existence of another node between them. Similarly, \( p \) checks if \( n \) is indeed its successor, and if not, it notifies \( n \). If either of \( s \) or \( p \) does not respond, \( n \) pings it repeatedly until it times out. The lack of response means that the node is either unreachable or dead. This leads to the update of the successor and predecessor pointers in the vicinity of that node, following the Chord protocol, as well as to a global propagation of the leave event, which we describe next.

A joining node contacts another system node to perform a lookup to its own ID, enabling it to get in touch with its predecessor and successor, thus informing them of its presence.

To maintain correct full routing tables, notifications of membership change events, i.e., joins and leaves, must reach every node in the system within a specified amount of time (depending on what fraction of failed queries, i.e., \( f \), is deemed acceptable). Our goal is to do this in a way that has low notification delay yet reasonable bandwidth consumption, since bandwidth is likely to be the scarcest resource in the system.

We achieve this goal by superimposing a well-defined hierarchy on the system. This hierarchy is used to form dissemination trees, which are used to propagate event information. Our goal is to ensure that each event is transmitted to a particular node only once, thus avoiding bandwidth waste. In Section 3.2, we will see that as the membership changes, duplications may occur, but they are rare.

We impose this hierarchy on a system with dynamic membership by dividing the \( m \)-bit circular identifier space into \( k \) equal contiguous intervals called slices. The \( i \)th slice contains all nodes currently in the overlay whose node identifiers lie in the range \([i \cdot 2^m/k, (i + 1) \cdot 2^m/k)\). Since nodes have uniformly distributed random identifiers, these slices will have about the same number of nodes at any time. Each slice has a slice leader, which is chosen dynamically as the node that is the successor of the midpoint of the slice identifier space. For example, the slice leader of the \( i \)th slice is the successor node of the key \((i + 1/2) \cdot 2^m/k\).

2. A more recent study [8] shows a slightly longer mean session time of 4.5 hours, but their experimental methodology removes peers with very short lifetimes from the sample.
When a new node joins the system, it learns about the slice leader from one of its neighbors along with other information like the data it is responsible for and its routing table.

Additionally, each slice is divided into \( u \) equal-sized intervals called units. Each unit has a unit leader, which is dynamically chosen as the successor of the midpoint of the unit.

Fig. 1 depicts how information flows in the system. When a node (labeled \( X \) in Fig. 1) detects a change in membership (its successor failed or it has a new successor), it sends an event notification message to its slice leader. The slice leader collects all event notifications it receives from its own slice and aggregates them for \( t_{\text{big}} \) time units before sending a message to other slice leaders. To spread out bandwidth utilization, communication with different slice leaders is not synchronized: the slice leader ensures only that it communicates with each individual slice leader once every \( t_{\text{big}} \). Therefore, messages to different slice leaders are normally sent at different points in time.

The slice leaders aggregate messages they receive for a short time period \( t_{\text{wait}} \) and then dispatch the aggregate message to all unit leaders of their respective slices. A unit leader piggybacks this information on its keep-alive messages to its successor and predecessor.

Other nodes propagate this information in one direction: if they receive information from their predecessors, they send it to their successors and vice versa. The information is piggybacked on keep-alive messages. In this way, all nodes in the system receive notification of all events, but within a unit, information is always flowing from the unit leader to the ends of the unit. Nodes at unit boundaries do not send information to their neighboring nodes outside their unit. As a result, there is no redundancy in the communications: a node will get information only from its neighbor that is one step closer to its unit leader.

We get several benefits from choosing this design. First, it imposes a structure on the system, with well-defined event dissemination trees. This structure helps us ensure that there is no redundancy in communications, which leads to efficient bandwidth usage.

Second, the aggregation of several events into one message allows us to avoid small messages, particularly within a slice. Small messages are a problem since the protocol overhead becomes significant relative to the message size, leading to higher bandwidth usage. This effect will be analyzed in more detail in Section 4.

### 3.2 Fault Tolerance

If a query fails on its first attempt, it does not return an error to an application. Instead, queries can be rerouted. If a query from node \( n_1 \) to node \( n_2 \) fails because \( n_2 \) is no longer in the system, \( n_1 \) can retry the query by sending it to \( n_2 \)'s successor. If the query failed because a recently joined node, \( n_3 \), is the new successor for the key that \( n_1 \) is looking up, \( n_2 \) can reply with the identity of \( n_3 \) (if it knows about \( n_3 \)), and \( n_1 \) can contact \( n_3 \) in a second routing step.

Our scheme is dependent on the correct functioning of slice leaders, so we need to recover from their failure. Since there are relatively few slice leaders, their failures are infrequent. Therefore, we do not have to be very aggressive about replacing them in order to maintain our query success target. When a slice or unit leader fails, its successor soon detects the failure and becomes the new leader.

Between the time a slice or unit leader fails and the time a new node takes over, some event notification messages may be lost, and the information about those membership changes will not be reflected in the system nodes’ membership tables. This is not an issue for routing correctness, since each node maintains correct information about its predecessor and successor. It will, however, lead to more routing hops, and if we allowed these errors to accumulate, it would eventually lead to a degradation of the one-hop lookup success rate.

To avoid this accumulation, we use the lookups themselves to detect and correct these inaccuracies. When a node performs a lookup and detects that its routing entry is incorrect (i.e., the lookup timed out or was rerouted to a new successor), this new information is then pushed to all the system nodes via the normal channels: it notifies its slice leader about the event. This can lead to the transmission of redundant information throughout the system, but our results show that this case is sufficiently infrequent to be considered acceptable.

The correctness of our protocols is based on the fact that successor and predecessor pointers are correct. This ensures that even if the remainder of the membership information contains errors, the query will eventually succeed after rerouting. In other words, our complete membership description can be seen as an optimization to following successor pointers, in the same way as Chord fingers are an optimization to successors (or similarly, for other peer-to-peer routing schemes). Furthermore, we can argue that our successor and predecessor pointers are correct due to the fact that we follow essentially the same protocol as Chord to maintain these, and this has already been proven correct.

### 3.3 Scalability with System Size

Slice leaders have more work to do than other nodes, and this might be a problem for a poorly provisioned node with a low-bandwidth connection to the Internet. To overcome this problem, we can identify well-connected and well-provisioned nodes as “supernodes” on entry into the system (as in [12]). There can be a parallel ring of supernodes, and the successor (in the supernode ring) of the midpoint of the slice identifier space becomes the slice leader. In the case of not having enough supernodes in the system, we could elect leaders from nonsupernodes in a subset of the slices.

As we will show in Section 4, bandwidth requirements are small enough to make most participants in the system...
potential supernodes in a $10^3$-sized system (in such a system, slice leaders will require 35-Kbps upstream bandwidth). In a million-node system, we may require supernodes to be well-connected academic or corporate users (the bandwidth requirements increase to 350 Kbps).

3.4 Sparse Network

Our protocol is configured with a static number of slices and units, and therefore, it needs to take into account the initial stages of the system where the total number of nodes is small. In a sparsely populated ring, there is a chance of there being no nodes between the midpoint and the end of the slice or unit, and so, in these cases, the successor would be outside the slice or unit that it would lead. There is also the possibility that a single node could be elected leader of two or more different slices or units or elected both a slice and a unit leader. Therefore, we add the requirement that if such a node does not exist within the slice or unit, then the leader is the predecessor of the midpoint if that node is contained in the slice or unit, respectively. If a slice or unit is not populated, a leader will only be assigned when the first node joins the slice or unit. We also considered an alternative adaptative design, where slices start out big (e.g., the whole ring) and then split or merge as the system size evolves. We abandoned this design because it introduced the additional complexity of keeping a coherent view of the hierarchy across system nodes.

4 Analysis

This section presents an analysis of how to parameterize the system to satisfy our goal of fast propagation. To achieve our desired success rate, we need to propagate information about events within some time period $t_{tot}$; we begin this section by showing how to compute this quantity. Yet we also require good performance, especially with respect to bandwidth utilization. Later in the section, we show how we satisfy this requirement by controlling the number of slices and units.

Our analysis considers only nonfailure situations. It does not take into account overheads of slice and unit leader failures because these events are rare. It also ignores message loss and delay since this simplifies the presentation, and the overhead introduced by message delays and retransmissions is small compared to other costs in the system.

Our analysis assumes that query targets are distributed uniformly throughout the ring. It is based on a worst-case pattern of events, queries, and notifications: we assume all events happen just after the last slice-leader notifications, and all queries happen immediately after that, so that none of the affected routing table entries has been corrected, and all queries targeted to those nodes (i.e., the nodes causing the events) fail. In a real deployment, queries would be interleaved with events and notifications, so fewer of them would fail.

This scenario is illustrated by the timeline in Fig. 2. Here, $t_{wait}$ is the time slice leaders wait to communicate with their unit leaders, $t_{small}$ is the time it takes to propagate information throughout a unit, and $t_{big}$ is the time a slice leader waits between communications to some other slice leader. Within $t_{wait} + t_{small}$ (point 4), slices in which the events occurred all have correct entries for nodes affected by the respective events. $t_{big}$ units of time after the events (point 5), slice leaders notify other slice leaders. Within a further $t_{wait} + t_{small}$ (point 7), all nodes in the system receive notification about all events.

Thus, $t_{tot} = t_{detect} + t_{wait} + t_{small} + t_{big}$. The quantity $t_{detect}$ represents the delay between the time an event occurs and the time when the leader of that slice first learns about it.

4.1 Configuration Parameters

The following parameters characterize a system deployment:

1. $f$ is the acceptable fraction of queries that fail in the first routing attempt.
2. $n$ is the expected number of nodes in the system.
3. $r$ is the expected rate of membership changes in the system.

Given these parameters, we can compute $t_{tot}$. Our assumption that query targets are distributed uniformly around the ring implies that the fraction of failed queries is proportional to the expected number of incorrect entries in a querying node’s routing table. Given our worst-case assumption, all the entries concerning events that occurred in the last $t_{tot}$ units of time are incorrect, and therefore, the fraction of failed queries is $\frac{t_{tot}}{n}$. Therefore, to ensure that no more than a fraction $f$ of queries fail, we need

$$t_{tot} \leq \frac{f \times n}{r}.$$

For a system with $10^6$ nodes, with a rate of 200 events/s and $f = 1$ percent, we get a time interval as large as 50 seconds to propagate all information. Note also that if $r$ is linearly proportional to $n$, then $t_{tot}$ is independent of $n$. It is only a function of the desired success rate.

4.2 Slices and Units

Our system performance depends on the number of slices and units:

1. $k$ is the number of slices the ring is divided into.
2. $u$ is the number of units in a slice.

Parameters $k$ and $u$ determine the expected unit size. This in turn determines $t_{small}$, the time it takes for information to propagate from a unit leader to all members of a unit, given an assumption about $h$, the time between keep-alive probes. From $t_{small}$, we can determine $t_{big}$ from our calculated value for $t_{tot}$, given choices of values for $t_{wait}$ and $t_{detect}$. (Recall that $t_{tot} = t_{detect} + t_{big} + t_{wait} + t_{small}$.)

To simplify the analysis, we will choose values for $h$, $t_{detect}$, and $t_{wait}$. As a result, our analysis will be concerned with just two independent variables, $k$ and $u$, given a particular choice of values for $n$, $r$, and $f$. We will use 1 second for both $h$ and $t_{wait}$. This is a reasonable decision...
TABLE 1
Summary of Bandwidth Use

<table>
<thead>
<tr>
<th></th>
<th>Upstream</th>
<th>Downstream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice Leader</td>
<td>$\frac{r \cdot v}{k} + \left( \frac{r \cdot m}{k} + \frac{2 \cdot v}{t_{big}} \right) \cdot (k - 1) + (r \cdot m + v) \cdot u$</td>
<td>$\frac{r \cdot (m + v)}{k} + \left( \frac{r \cdot m}{k} + \frac{2 \cdot v}{t_{big}} \right) \cdot (k - 1) + u \cdot v$</td>
</tr>
<tr>
<td>Unit Leader</td>
<td>$2(r \cdot m + v) + v$</td>
<td>$r \cdot m + 3 \cdot v$</td>
</tr>
<tr>
<td>Other nodes</td>
<td>$r \cdot m + 2 \cdot v$</td>
<td>$r \cdot m + 2 \cdot v$</td>
</tr>
</tbody>
</table>

since the amount of data being sent in probes and messages to unit leaders is large enough to make the overhead in these messages small (e.g., information about 20 events will be sent in a system with $10^8$ nodes). Note that with this choice of $h$, $t_{small}$ will be half the unit size. We will use 3 seconds for $t_{detect}$ to account for the delay in detecting a missed keep-alive message and a few probes to confirm the event.

4.3 Cost Analysis
Our goal is to choose values for $k$ and $u$ in a way that reduces bandwidth utilization. In particular, we are concerned with minimizing bandwidth use at the slice leaders, since they have the most work to do in our approach.

To compute the optimal values for $k$ and $u$, we analyze the bandwidth utilization for each type of node in the system. Our analysis assumes that $m$ bytes are required to describe an event, and the overhead per message is $v$.

4.3.1 Slice Leaders
Slice leaders have three forms of communication:

- **Event notification to slice leaders.** Whenever a node detects an event, it sends a notification to its slice leader. The expected number of events per second in a slice is $r$. The downstream bandwidth utilization on slice leaders is therefore $\frac{r \cdot (m + v)}{k}$. Since each message must be acknowledged, the upstream utilization is $\frac{r \cdot v}{k}$.

- **Messages exchanged between slice leaders.** Each message sent from one slice leader to another batches together events that occurred in the last $t_{big}$ units of time in the slice. The typical message size is therefore $r \cdot t_{big} \cdot m + v$ bytes. During any $t_{big}$ period, a slice leader sends this message to all other slice leaders ($k - 1$ of them) and receives an acknowledgment from each of them. Since each slice leader receives as much as it sends on the average, the upstream and downstream use of bandwidth is symmetric. Therefore, the bandwidth utilization (both upstream and downstream) is

$$\left( \frac{r \cdot m}{k} + \frac{2 \cdot v}{t_{big}} \right) \cdot (k - 1).$$

- **Event propagation to unit leaders.** Slice leaders also send messages to unit leaders and handle the respective acknowledgments. Messages received by a slice leader are batched for 1 second before being forwarded to unit leaders. In 1 second, $r$ events happen and, therefore, the aggregate message size is $(r \cdot m + v)$, and the upstream bandwidth utilization is

$$(r \cdot m + v) \cdot u.$$

In terms of downstream bandwidth, slice leaders handle $u$ acknowledgments every second, which corresponds to a bandwidth of $u \cdot v$.

4.3.2 Unit Leaders
Unit leaders handle the messages from slice leaders, which corresponds to a downstream bandwidth utilization of

$$r \cdot m + v$$

and an upstream bandwidth for acknowledgments of $v$.

These messages are forwarded to both their predecessor and successor. This corresponds to an upstream bandwidth of twice the cost of the slice-leader-to-unit-leader communication or

$$2(r \cdot m + v).$$

Again, we have to consider the acknowledgments that cost $2 \cdot v$.

4.3.3 Other Nodes
The remaining nodes exchange keep alives, which form the base-level communication between a node and its predecessor and successor. These messages include information about recent events.

Since keep-alive messages are sent every second, every node that is not on the edge of a unit will send and acknowledge an aggregate message containing, on the average, $r$ events. The size of this message is therefore $r \cdot m + v$, and the size of the acknowledgment is $v$.

Ordinary nodes that are on the edge of the units do not need to further propagate the events. However, they have the same download demands as the other ordinary nodes. Since these represent a small fraction of the ordinary nodes, in the rest of the analysis, we will assume that all ordinary nodes need to relay the events.

4.3.4 Summary
Table 1 summarizes the net bandwidth use on each node.

Using these formulas, we can compute the bandwidth utilization on nonslice leaders in a particular configuration. In these computations, we use $m = 10$ bytes and $v = 20$ bytes. In a system with $10^7$ nodes and $r = 20$ events/s, we see that the total load on an ordinary node is 3.84 Kbps (0.47 Kbytes/s) and the load on a unit leader is 3.68 Kbps (0.45 Kbytes/s).
upstream and 2.08 Kbps (0.25 Kbytes/s) downstream. For a system with $10^6$ nodes, these numbers become 32.6 Kbps (3.98 Kbytes/s), 32.5 Kbps (3.96 Kbytes/s), and 16.5 Kbps (2.01 Kbytes/s), respectively.

In the table, it is clear that the upstream bandwidth required for a slice leader is likely to be the dominating and limiting term. Therefore, we shall choose parameters that minimize this bandwidth. By simplifying the expression and using the interrelationship between $u$ and $t_{big}$ (explained in Section 4.2), we get a function that depends on two independent variables $k$ and $u$. By analyzing the function, we deduce that the minimum is achieved for the following values:

$$k = \sqrt{\frac{r \cdot m \cdot n}{4 \cdot v}},$$

$$u = \sqrt{\frac{4 \cdot v \cdot n}{r \cdot m \cdot (t_{tot} - t_{wait} - t_{detect})^2}}.$$

These formulas allow us to compute values for $k$ and $u$. For example, in a system of $10^5$ nodes, we want roughly 500 slices, each containing five units. In a system of $10^6$ nodes, we still have five units per slice, but now, there are 5,000 slices.

Given values for $k$ and $u$, we can compute the unit size, and this in turn allows us to compute $t_{small}$ and $t_{big}$. We find that we use the least bandwidth when

$$t_{small} \approx t_{big}.$$

Thus, we choose 23 seconds for $t_{big}$ and 23 seconds for $t_{small}$.

Given these values and the formulas in Table 1, we can plot the bandwidth usage per slice leader in systems of various sizes. The results of this calculation are shown in Fig. 3. Note that the load increases only linearly with the size of the system. The load is quite modest in a system with $10^5$ nodes (17-Kbps upstream bandwidth), and therefore, even nodes behind cable modems can act as slice leaders in such a system. In a system with $10^6$ nodes, the upstream bandwidth required at a slice leader is approximately 166 Kbps. Here, it would be more appropriate to limit slice leaders to being machines on reasonably provisioned local area networks. For larger networks, the bandwidth increases to a point where a slice leader would need to be a well-provisioned node.

Fig. 4 shows the percentage overhead of this scheme in terms of aggregate bandwidth used in the system with respect to the hypothetical optimum scheme with zero overhead. In such a scheme, the cost is just the total bandwidth used in sending $r$ events to every node in the system every second, i.e., $r \cdot n \cdot m$. Note that the overhead in our system comes from the per-message protocol overhead. The scheme itself does not propagate any redundant information. We note that the overhead is approximately 19 percent for a system containing $10^5$ nodes and goes down to 4 percent for a system containing $10^6$ nodes. This result is reasonable because messages get larger and the overhead becomes less significant as the system size increases.

### 4.4 Discussion

In this section, we analyze the limitations of OneHop; in particular, we discuss its limits in terms of scalability and what can be done when these limits are reached.

The two main issues that hamper scalability are memory usage and network traffic. In terms of memory requirements, keeping complete membership information requires about 26 bytes per node (assuming that node IDs are based on the SHA-1 function that outputs 160 bits and assuming that network addresses consist of an IPv4 address and a port number). Therefore, assuming that 100 Mbytes is a reasonable amount of memory to be dedicated to this table, we can scale to four million nodes.

As far as bandwidth usage is concerned and assuming a worst-case scenario where slice leaders are behind a residential 100-Kbps uplink, our algorithm stops working at around 300,000 nodes, according to the previous analysis.

After this point, we probably want to move to a multihop overlay. The natural next step would be to use a two-hop algorithm such as the one we presented together with our original publication of OneHop [1]. Our analysis in that paper showed that it could scale to very large systems, even with tens of millions of nodes, which is the current scale for the largest peer-to-peer systems that are deployed.

Despite the scalability of a two-hop algorithm, we also analyzed the alternatives if we needed to support an even larger number of nodes. We considered the possibility of generalizing the two-hop protocol to obtain a $k$-hop system but decided to reject that solution because generalizing the two-hop system to support an arbitrary number of hops turned out to be complex, and so, it makes more sense to use a traditional overlay like Pastry [4], which can already get close to achieving three or four hops (which would be the new hop counts a $k$-hop scheme would enable). This can
be achieved both by caching and by appropriately choosing system parameters like the base used by the protocol.

5 SIMULATION RESULTS

In this section, we present experimental results obtained with simulations of OneHop. In the first set of experiments, we used a coarse-grained simulator to understand the overall behavior of OneHop with tens of thousands of nodes. This simulation scales to approximately 20,000 nodes. In the second set of experiments, we use a more fine-grained simulation of OneHop where the simulation environment could not support more than 2,000 nodes.

In both experiments, we derived interhost latencies from Internet measurements among a set of 1,024 DNS servers [13]. Note that our experimental results are largely independent of topology since we did not measure lookup latency, and the impact of internode latency on query failure rates is minimal, since the latencies are over an order of magnitude smaller than the time-outs used for propagating information.

In evaluating our protocol, we focus on measuring the number of lookups that succeed in a single hop instead of measuring latencies. Using absolute latencies of lookups as an evaluation metric would provide results that are highly dependent on the topology. Instead, by measuring that rate of lookups that succeed in a single hop (which is the ideal situation), we capture clearly the benefits of the OneHop protocol. Furthermore, given that we expect most lookups to complete in a single hop, this means that our protocol will have a stretch (relative to IP) very close to 1. This will automatically allow for meaningful comparisons with any other protocol that also measures stretch: such value will also automatically allow for meaningful comparisons with any other protocol that also measures stretch.

5.1 Coarse-Grained Simulations

The experiments using the coarse-grained simulator were aimed at validating our analytic results concerning the query success rate. The coarse-grained simulator is based on some simplifying assumptions that allow it to scale to larger network sizes. First, it is synchronous: the simulation proceeds in a series of rounds (each representing 1 second of processing), where all nodes receive messages, perform local processing of the messages, and send messages to other nodes. Second, neither slice-leader failures nor packet losses were simulated.

The first set of experiments shows the fraction of successful queries as a function of the interslice communication frequency. The expected number of nodes in the system is 20,000, the mean join rate is two nodes per second, and the mean leave rate is two nodes per second. Node lifetime is exponentially distributed. New nodes and queries are distributed uniformly in the ID space. The query rate is 1 query per node per second.

The number of slices is chosen to be 50, and there are five units in every slice (these are appropriate choices according to our analysis). The frequency of interslice communication varied from one every 10 seconds to one every 100 seconds.

The results are shown in Fig. 5. We can see that the query failure rate grows steadily with the time between interslice communications. Note, however, that even with a relatively infrequent communication of once every 100 seconds, we still obtain an average of about 1 percent failure rate.

This simulation confirms our expectation that the failed query rate we computed analytically was conservative. We can see that when the interslice communication is set to 23 seconds (the value suggested by our analysis), the query failure rate is about 0.4 percent and not 1 percent as we conservatively predicted in Section 3. The reason why the actual failure rate is lower is because our analysis assumed the worst case where all queries are issued right after membership events occur and before any events were propagated. In reality, queries are distributed through the time interval that it takes to propagate the information, and by the time some queries are issued, some nodes already have received the most up-to-date information.

5.2 Fine-Grained Simulations

In this section, we report on simulations of the one-hop routing algorithm using p2psim [14], a peer-to-peer protocol simulator where we could implement the complete functionality of the one-hop protocol. Using this simulator, we are able to explore bandwidth utilization and also the impact of slice and unit-leader failures.

In the first experiment, we measure the evolution of the query failure rate of a system that grows very fast initially and then stabilizes to “typical” membership dynamics. We simulate a system with 2,000 dynamic nodes with 10 slices and five units per slice. The results are shown in Fig. 6. In the first 300 seconds of the simulation, all nodes join rapidly. After that the system shows Gnutella-like churn [6] with 24 events per minute. All nodes join by obtaining a routing table from another node; the routing tables then continue to grow as new nodes come in.
After approximately the first 10 minutes, the query failure rate stayed consistently at around 0.2 percent. We also did experiments to determine the failure rate observed after the query is rerouted once. In this case, the failure rate settles down to approximately one failure in $10^4$ queries or 0.01 percent. This is because the local information about a slice, especially knowledge of predecessors and successors, gets transmitted very rapidly. Thus, 99.99 percent of the queries are correctly satisfied by two or fewer hops.

Next, we examine the behavior of the system in the presence of a burst of membership departures. Again, we simulated a system with 2,000 nodes, with 10 slices and five units per slice. The query rate was 1 lookup per second per node. At time instant $t = 8,000$ seconds, 45 percent of the nodes crash at $t = 8,000$ seconds. These nodes were chosen randomly. Fig. 7 shows the fraction of lookups that failed subsequent to the crash. It takes the system about 50 seconds to return to a reasonable query success rate, but it does not stabilize at the same query success rate that it had prior to the crash for another 350 seconds. What is happening in this interval is recovery from slice-leader failures. The query rate has an important role in slice-leader recovery; queries help in restoring the stale state by regenerating event notifications that were lost because of slice-leader failures.

Fig. 8 shows the overall bandwidth used in the system in this period. The aggregate bandwidth used in the entire system is around 300 Kbps before the crash and settles into around 180 Kbps after approximately 150 seconds. (The steady-state bandwidth decreases due to the decrease in the system size.) We can see that while the duration of the spike is similar to that of the spike in query failure rate, the bandwidth settles down to expected levels faster than successful lookups. This happens because lookups continue to fail on routing table entries whose notifications are lost in slice-leader failures. These failures have to be “rediscovered” by lookups and then fixed slice by slice, which takes longer. While each failed lookup generates notifications and thus increases maintenance bandwidth, at the system size of around 2,000, most messages (after the spike settles down) are dominated by the UDP/IP packet overhead. Thus, the overall effect on bandwidth is significantly lower.

We also ran experiments in which we simulated bursts of crashes of different fractions of nodes. We observed that the time periods taken for the lookups and bandwidth to settle down were almost the same in all cases. We expect this to happen because the time taken to stabilize in the system is dependent on the system size and chosen parameters of unit size and $t_{big}$, which remain the same in all cases.

We also computed the average spike bandwidth. This was measured by computing the average bandwidth used by the entire system in the 50 seconds it took for the system to settle down in all cases. In Fig. 9, we see that the bandwidth use grows approximately linearly with the size of the crash. In all cases, the bandwidth consumption is reasonable, given the fact that this bandwidth is split among over a thousand nodes.

6 IMPLEMENTATION
We have implemented the OneHop as an extension to the code developed for the Chord project [9]. This section describes the changes we made to the Chord code and some design details to make OneHop more practical.

6.1 Software Changes
The code base for Chord is written in C++ and makes use of the libasync library to manage communications asynchronously [15].

The Chord project is organized into several layers of code. The first layer is the routing substrate that implements the $find_successor()$ primitive. On top of that layer, there is the $dhash$ layer, which implements a distributed hash table (DHT) with a $get/put$ interface. Finally, the project includes some applications that are built on top of the $dhash$ layer, such as CFS [16], UsenetDHT [17], and OverCite [18].
Applications interact with Chord nodes to store or retrieve data by communicating with a local Location Service Daemon (lsd) process, which provides the put and get primitives. Communication between the applications and the lsd process is done using a Unix domain socket. The lsd process in turn communicates with the other lsd processes across the network (i.e., other nodes in the system) to exchange routing messages or to store and retrieve data blocks. The lsd process implements the client and server functionalities of both the routing and the DHT layer.

The Chord code implements different routing protocols that can be used by the routing layer (e.g., Chord [2] or de Bruijn [19]). When starting the lsd process, we can specify which protocol to use using a command line flag. This allowed us to extend the Chord code to support the OneHop protocol without removing the existing routing protocols supported by the system.

Each lsd process might run several virtual nodes. Running multiple nodes on the same machine can be useful for load balancing or debugging reasons. As such, according to the user parameters, the specified number of virtual nodes are created during the process initialization.

Each virtual node is implemented by an instance of a subclass of the vnode interface. Different subclasses of vnode are responsible for implementing different routing protocols. During initialization, the lsd process calls a vnode factory method according to the flags selected by the user.

Our implementation adds about 2,600 lines of code to the routing layer, and most of our code is divided over two new classes: onehop and route_onehop. The onehop class inherits from the existing vnode_impl class (which inherits from vnode) and is responsible for the overall implementation and coordination of the multiple aspects of the protocol. This class uses some of the functionalities (such as the management of the successor and predecessor pointers) of the vnode_impl class that implements the Chord protocol without the use of the fingers table.

Our onehop class overrides the join and the dispatch methods of the vnode_impl class. The join method contacts the bootstrapping node to find the successor of the incoming node and then asks it for a copy of the routing table. The dispatch method is the handler for incoming messages, which we changed to handle new RPCs that we defined for our protocol.

The route_onehop class is responsible for implementing lookups using the OneHop protocol and inherits from the class route_chord that implements the lookup using the standard Chord lookup scheme. It was not possible to use the existing route_chord without modification because the lookup protocol used in Chord first finds the closest predecessor for the identifier, which then outputs the responsible node. Therefore, using this class without modifying this protocol would require contacting at least two nodes for each lookup instead of one as is intended in the OneHop protocol.

We also created a new class (onehop_events_list) that is responsible for managing the event queues. Since the OneHop protocol is time driven, this class also keeps track of time.

### 6.2 Practical Design Choices

As we implemented and tested the OneHop system, we discovered ways to improve our protocol performance and robustness. In this section, we describe these improvements.

**Ordering of events.** In our implementation, we were careful to avoid changing the order in which events are handled. For example, there could be a leave event immediately followed by a join referring to the same node due to temporary unreachability. If these two events are processed in the wrong order, the node would mistakenly be removed from the routing tables of other nodes. To handle this problem, we manage retransmissions at the application level, and the retransmission of RPCs that carry event lists is done in a manner that preserves the correct ordering of events.

**Avoiding false events.** We noticed in our tests that generating a join event every time the successor pointer of a node changed, due to a joining event of a node closer than the previous successor, would generate two join events instead of one each time a new node joined the overlay. This would happen because the predecessor rightfully detects the joining node, but the joining node detects that it has a new successor (before nodes actually join, their successor points to themselves). To solve this problem we created a new message that asks a node’s successor if its entrance has already been signaled. Each node only answers affirmatively once for every session. This assures that the joining event is always signaled when it would be by just detecting the change in successor but only the first time for every node. This procedure does not affect the generation of new events when lookups fail.

In general, we realized that our implementation became more robust by carefully verifying the validity of an event before generating it (e.g., by attempting to contact a node more than once before declaring it unreachable or by using the aforementioned RPC that avoids generating more than one join event per node), since a misfired event has a big impact on system performance.

**Avoiding loops.** In some units, the last node of a unit might be a slice leader instead of an ordinary node. In this case, when ordinary nodes resend the events along the unit, from the unit leader to the end points of the unit, they must be careful so that the events are not mistakenly passed to the slice leader (because it could interpret them as being new events and therefore spread them throughout the network again). This problem could have been solved by creating distinct messages for these two situations. But instead, we chose a more extensible approach, where we added two fields to the message that specify what type of nodes the sender thinks the destination and the source (itself) are. The destination node will then discard the events that are not meant for it, according to the field indicating the expected destination node type. This mechanism does not increase the bandwidth usage by much since these fields are used only for event lists (i.e., not for every event in the list).

**Routing table downloads.** Another implementation choice was related to how joining nodes download the routing table from another node to initialize its own table. To improve load balancing, the joining node will look up its
successor in the ring and ask it for the routing table instead of pulling the routing table from the bootstrap node since uploading the routing table is a bandwidth-intensive operation and the distribution of bootstrap nodes might not be uniform.

**Changing node types.** Finally, we also addressed the issue of what to do with messages that are buffered at a node when a node changes its status. In our implementation, each node maintains a queue of events that are scheduled to be sent to the other nodes with which it communicates. When a node changes from one type to another, its new role may no longer require it to forward those messages. For instance, a slice leader maintains queues of events to send to the leaders of the other slices, but if it is downgraded to an ordinary role due to the join of a new node, it does not have to communicate with the other slice leaders anymore. Intuitively, the best solution might be to flush the buffered messages before downgrading the status of the node. However, we found that in some circumstances, this would create a cycle of events. Therefore, we chose to clear the queues by discarding existing events. This is one of the occasions where events might get lost, but by learning from lookups, such losses are eventually corrected.

### 6.3 Implementation Evaluation

In this section, we evaluate our implementation experimentally. The goal of this evaluation is to answer two main questions: 1) were the simulation results a valid approximation of a real deployment and 2) how effective is our implementation of OneHop? We do not compare ourselves to more recent proposals that build upon our work, since most of them have only been evaluated through simulation, which makes a comparative evaluation impossible. However, we expect these protocols to outperform OneHop on some vectors (such as load balancing) since they build upon our work.

We evaluated our implementation using ModelNet [10] to emulate the latency between nodes. We measured the lookup failure rate using the conservative definition of failure (lookups requiring more than one hop), and we also measured bandwidth usage. Our tests were conducted on a cluster with Pentium IV 3.2-GHz PCs, where up to 50 instances of our implementation were executed in each machine. Another machine (with Pentium III 1-GHz processor) was used to emulate the network core with the aid of ModelNet components.

Our emulated network had a total of 8,000 virtual core nodes and 2,000 edge nodes distributed over 500 stubs. The core nodes emulate the Internet routers, while the edge nodes emulate end points (where the application instances may run) [10]. Nodes were connected with 1,000-Kbps symmetrical links, and the default values were used for the rest of ModelNet parameters.

In our experiments, the mean rate of joins was one for every 10,000 nodes per second, and so was the rate of leaves. Both kinds of events were exponentially distributed. We used a constant query rate of 1 query per node every second, and queries were distributed uniformly over the identifier space.

To evaluate the impact of interslice communication, we ran the system with different periods of communication between slice leaders. In this test, we used 400 nodes, distributed over 64 slices, each with two units. We then calculated the mean query failure rate over five runs for each configuration setting. Fig. 10 clearly shows an increase of the query failure rate as the period between communications among slice leaders increases.

To perform this test, we had to configure the system with a significant number of slices even though it created slices with a small number of nodes. Otherwise, if the slices were to represent a large fraction of the ring, the importance of interslice communication would be reduced since the intraslice notifications would suffice to propagate information to a large fraction of system nodes.

Fig. 11 shows the query failure rate over time in a system with 600 nodes. The network was divided into eight slices each with two units. During the first 500 seconds, the nodes quickly join the network and, subsequently, the standard event rate is adopted. After the initial starting period completes, we observed that the query failure rate quickly stabilized to approximately 1 percent. During the rest of the experiment, the query failure rate remained near or below 1 percent.

The other experiment consisted of abruptly crashing a large fraction (45 percent) of the system nodes to evaluate how well the protocol handles network partitions or extreme events. Once again, we ran the system with 600 nodes and eight slices. When $t = 3,600$ seconds, we crashed 45 percent of the running instances. The rate of events per node per second during the experiment was constant.

Fig. 12 shows the query failure rate evolution under these circumstances. When the nodes crash, a spike of query failure rates occurs. The system then recovers to rates lower than 4 percent in the next 200 seconds.
algorithms is quite different—Pastry uses only \( O \) substantially from ours in that the nature of the routing algorithm in a dynamic setting. This work differs first attempt.

Fig. 12. Query failure rate in a system with 600 nodes when 45 percent of the system nodes crash at \( t = 3,600 \) seconds.

In the same experiment, we also measured the overall bandwidth usage. These measurements reflect the sum of the upload bandwidth (which is different from the download bandwidth since packets could get lost) of all nodes in the system over time. The results presented in Fig. 13 take into account Chord stabilization messages, event notifications and lookup messages. When the nodes crashed, we observed a surge in the overall bandwidth usage that reached about 9,000 Kbps. After approximately 200 seconds, the bandwidth usage was reduced to values lower than the initial ones.

Fig. 13. Overall bandwidth usage in a system with 600 nodes when 45 percent of the system nodes crash at \( t = 3,600 \) seconds.

7 Related Work

Rodrigues et al. [20] proposed a single-hop DHT, but they assumed much smaller peer dynamics, like those in a corporate environment, and therefore did not have to deal with the difficulties of rapidly handling a large number of membership changes with efficient bandwidth usage. Douceur et al. [21] present a system that routes in a constant number of hops, but that design assumes smaller peer dynamics, and searches can be lossy.

Kelips [22] uses \( \sqrt{n} \)-sized tables per node and a gossip mechanism to propagate event notifications to provide constant-time lookups. Their lookups, however, are constant time only when the routing table entries are reasonably accurate. As seen before, these systems are highly dynamic, and the accuracy of the tables depends on how long it takes for the system to converge after an event. The expected convergence time for an event in Kelips is \( O(\sqrt{n} \times \log(\log n)) \). While this will be tens of seconds for small systems of around 1,000 nodes, for systems having \( 10^5 \) to \( 10^6 \) nodes, it takes over an hour for an event to be propagated through the system. At this rate, a large fraction of the routing entries in each table is likely to be stale, and a correspondingly large fraction of queries would fail on their first attempt.

Mahajan et al. [23] also derive analytic models for the cost of maintaining reliability in the Pastry [4] peer-to-peer routing algorithm in a dynamic setting. This work differs substantially from ours in that the nature of the routing algorithms is quite different—Pastry uses only \( O(\log N) \) state but requires \( O(\log N) \) hops per lookup—and they focus their work on techniques to reduce their (already low) maintenance cost.

Liben-Nowell et al. [24] provide a lower bound on the cost of maintaining routing information in peer-to-peer networks that try to maintain a topological structure. We are designing a system that requires significantly larger bandwidth than in the lower bound because we aim to achieve a much lower lookup latency.

Accordion [25] proposes a system that adapts the size of its routing state in order to trade bandwidth for lookup efficiency. This work improves OneHop by allowing the routing substrate to adapt the number of hops it can deliver, according to the bandwidth the system can afford to spend.

Since we first published our OneHop protocol [1], a number of other one-hop protocols have been proposed [26], [27], [28]. These protocols also achieve full-information lookups but improve on the original OneHop design by not requiring the network to maintain a fixed event dissemination hierarchy and avoiding an asymmetric bandwidth consumption across nodes.

8 Conclusions

This paper questions the necessity of multihop lookups in peer-to-peer routing algorithms. We introduce a novel peer-to-peer lookup algorithm that routes in one hop, unless the lookup fails and other routes need to be attempted. We designed our algorithm to provide a small fraction of lookup failures (e.g., 1 percent).

We present analytic results that show how we can parameterize the system to obtain reasonable bandwidth consumption, despite the fact that we are dealing with a highly dynamic membership. We present simulation results that support our analysis that the system delivers a large fraction of lookups within one hop. We also implemented this protocol and our experimental evaluation in an emulated wide area network to confirm the simulation results.

The OneHop protocol is now available as an additional routing choice for the Chord peer-to-peer DHT implementation, allowing applications that used this DHT to be deployed using one-hop lookups, without the need to modify their code.

Acknowledgments

The authors are very grateful to Luiz Monnerat for pointing out some corrections to the analysis. They would like to thank their NSDI shepherd Ion Stoica, the anonymous referees, and everyone else who provided helpful comments on the contents of this paper. This research was supported by The US Defense Advanced Research Projects Agency (DARPA) under Contract F30602-98-1-0237 and by the US National
Science Foundation (NSF) under Cooperative Agreement ANI-0225660. Pedro Fonseca was supported by grants provided by INESC-ID and FCT.

REFERENCES


Pedro Fonseca received an undergraduate degree in computer science and engineering from the Instituto Superior Técnico (IST), Technical University of Lisbon (UTL), Portugal, in 2006. He is currently pursuing a PhD degree and is working at the Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern and Saarbrücken, Germany, where he leads the Dependable Systems Group. Previously, he was an assistant professor at IST and a researcher at INESC-ID. During his PhD, he was a researcher at the Computer Science and Artificial Intelligence Laboratory, MIT, under the supervision of Professor Barbara Liskov. His primary technical interest is in distributed systems, with a particular focus on large-scale distributed systems such as peer-to-peer overlays and fault tolerance. He has received several fellowships and awards, including a Best Paper Award at the 18th ACM Symposium on Operating Systems Principles (SOSP) and a special recognition award from the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Rodrigo Rodrigues received the undergraduate degree from the Instituto Superior Técnico (IST), Technical University of Lisbon, Portugal, in 1998 and the master's and PhD degrees from the Massachusetts Institute of Technology (MIT), Cambridge, in 2001 and 2005, respectively. He is a tenure-track faculty at the Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern and Saarbrücken, Germany, where he leads the Dependable Systems Group. Previously, he was an assistant professor at IST and a researcher at INESC-ID. During his PhD, he was a researcher at the Computer Science and Artificial Intelligence Laboratory, MIT, under the supervision of Professor Barbara Liskov. His primary technical interest is in distributed systems, with a particular focus on large-scale distributed systems such as peer-to-peer overlays and fault tolerance. He has received several fellowships and awards, including a Best Paper Award at the 18th ACM Symposium on Operating Systems Principles (SOSP) and a special recognition award from the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.
Anjali Gupta received the master’s degree from the Electrical Engineering and Computer Science Department, Massachusetts Institute of Technology (MIT), Cambridge. She was a member of the Programming Methodology Group in the Computer Science and Artificial Intelligence Laboratory (CSAIL), MIT. She is currently working in statistical genetics at the Wellcome Trust Centre for Human Genetics at the University of Oxford.

Barbara Liskov is a professor of computer science and engineering and the head of the Programming Methodology Group in the Computer Science and Artificial Intelligence Laboratory (CSAIL), Massachusetts Institute of Technology (MIT), Cambridge. Her research interests are in programming methodology, programming languages and systems, and distributed computing. Her major projects include the design and implementation of CLU, the first language to support data abstraction; the design and implementation of Argus, the first high-level language to support implementation of distributed programs; and the Thor object-oriented database system, which provides transactional access to persistent highly available objects in wide-scale distributed environments. Her current research interests include Byzantine fault-tolerant storage systems and efficient and secure online storage. She is a member of the National Academy of Engineering and a fellow of the American Academy of Arts and Sciences and the ACM. She received The Society of Women Engineers’ Achievement Award in 1996, the IEEE von Neumann medal in 2004, and the ACM SIGPLAN Programming Languages Achievement Award in 2008. She is a member of the IEEE.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.