Myrmic: Secure and Robust DHT Routing

Peng Wang, Nicholas Hopper, Ivan Osipkov and Yongdae Kim

ABSTRACT

A distributed hash table such as Chord attempts to build a persistent store from a network of (possibly unstable) peer nodes. There has been a great deal of work on making DHTs robust to environmental interference (such as membership churn, transient routing failures and high CPU load) but considerably less work on implementing DHTs that are secure against *adversarial* behavior designed to cause DHT failure. In this paper, we introduce *Myrmic*, a novel DHT routing protocol designed to be robust against adversarial interference. A key feature distinguishing Myrmic from other DHT *implementations* is a *root verification protocol* that allows anyone to verify that the node responding to a query for key k is indeed the "correct" holder of the key. We give analytical results showing that even when a large fraction of nodes, for example 30%, cooperate to adversarially interfere with query routing, Myrmic finds uncorrupted roots in expected logarithmic time, and confirm these results with simulations of 1000 nodes. Finally, we implement the proposed protocol and evaluate it through experimentation with 120 nodes on PlanetLab in order to measure wide area network performance. All of these results suggest that Myrmic provides stronger robustness guarantees while incurring minimal network and CPU overhead.

1. INTRODUCTION

A distributed hash table (DHT) is a service that maps *keys* in a flat identifier space onto *nodes* in a network of peers. Systems such as CAN [1], Chord [2], Pastry [3], OpenDHT [4] and Tapestry [5] structure peers into an overlay network such that each peer needs only remember $O(\log n)$ other peers and can locate any identifier in at most $O(\log n)$ hops. Because of their scalability, lack of a central point of failure, and design for fault tolerance, these systems have been used to construct a wide range of distributed applications, for example P2P file system [6], P2P archival systems [7, 8, 9], BitTorrent [10], P2P web cache systems [11, 12], P2P multicast systems [13, 14], and P2P DNS [15, 16].

Many of these DHT implementations have been engineered to tolerate faults caused by environmental conditions such as transient routing failures, overloaded CPUs, and membership churn. However, many of these systems do not deal with *adversarial* faults that maliciously prevent nodes from discovering the correct mapping between identifiers and peers. Interfering with this mapping can in turn invalidate the availability, correctness or security of protocols running on top of the DHT, since they assume the mapping to be available, correct and consistent. Since many of the systems proposed to be built on top of DHTs have direct financial or security implications, it is natural to expect that if they become popular, they will be targeted by adversaries who can control a significant fraction of nodes in the system.

Algorithmically, several DHT schemes that are provably robust to malicious failure have been proposed. These provably secure protocols are of interest because security proofs rule out *all possible future attacks* in addition to the set of currently known attacks. The literature on cryptographic network protocols has many examples of schemes, using strong cryptographic primitives, that were designed without a proof of security and eventually broken [17, 18, 19]. Unfortunately, while many of these provably secure DHT schemes scale well asymptotically (for example the scheme in [20] has latency $O(\log n)$ and bandwidth $O(\log^2 n)$) these parameters do not always translate well to implementations due to the constants involved. Thus these theoretical schemes, while interesting, are not practical for implementation.

In this paper we introduce and report on the PlanetLab [21] deployment of Myrmic, a DHT implementation with provable security against malicious node failures. Myrmic has the same semantics as Chord and in a network with no malicious nodes it has message cost and latency that are provably at most twice the cost of Chord with recursive routing. Our experiments with the system both in the wide-area PlanetLab testbed and in a local-area network show that good performance is maintained even when large fraction (for example, 30%) of nodes behave maliciously by dropping all routing requests. Thus, it can be used as a drop-in, secure replacement for Chord.

Three key ideas are involved in the design of Myrmic. First, we use a small set of trusted nodes to provide a kind of local admission control; these nodes store no state and may fail transiently with no effect on the security of the system. Second, our system is designed to tolerate failures on a small percentage δ of routing requests, while guaranteeing that these failures are transient, even when they are malicious. Third, we use a *root verification protocol* that with high probability allows only a single node to prove current ownership of a given key; this prevents many of the previously known attacks on overlay routing schemes. This root verification protocol is also directly applicable to other DHT routing protocols even if proximity neighbor selection [22] is used. To our knowledge, Myrmic is the only DHT protocol where root verification is *externally verifiable*: any node can check that the result of a lookup is correct. This simplifies some aspects of application design, for example, allowing new nodes to join the DHT or a client to use a generic DHT service such as OpenDHT[4] without the additional trust assumptions of a trusted gateway or the additional communication cost of using several redundant gateways. We use this verification protocol to augment the iterative routing protocol of Chord so that adversaries are constrained to either drop routing queries or give correct responses.

The remainder of this paper is organized as follows. Section 2 gives an overview of DHT protocols and Chord. Sections 3 and 4 give a more detailed overview of our threat model and the algorithms employed by Myrmic. We analyze the security of these algorithms briefly in section 5, and report on experimental evaluations in simulated, local-area and wide-area networks in section 6. Finally, we discuss related work in section 7.

2. BACKGROUND

2.1 Overview of DHT Routing

In this section, we briefly overview DHTs, using Chord [2] as a concrete example. DHT networks allow nodes to store and to retrieve data objects efficiently. Each node is assigned a unique identifier, or *nodeId*, and each application object is assigned a unique identifier, or *key*. Node identifiers are often computed as the cryptographic hash (e.g. SHA-1) of the node's public key or IP address, while a key is usually computed as the cryptographic hash of an application object's attributes, which can be used to identify the object. NodeIds and keys are uniformly distributed in the *Id space*, a set of n-bit integers. A key k is mapped to a unique node – the key's root is denoted as $root(k)$, based on numerical proximity. In Chord, the node $root(k)$ is the node with the smallest nodeId equal to or greater than k in the Id space. When a node inserts a key-value pair (k, v), $root(k)$ stores the pair, where the value v is application-specific information. When a node queries the key k, $root(k)$ returns v. In order to tolerate failures and/or expedite the query process, (k, v) can be replicated at several nodes, called *replica roots*. In Chord, the replica roots of (k, v) are $root(k)$ and its several successors.

A DHT provides a distributed lookup protocol, which allows queriers to communicate with the node that stores a particular data object efficiently. For this purpose, each node maintains a routing table containing a set of other nodes' nodeIds and IP addresses. The nodes in each routing table are chosen in such a way that a lookup message can be efficiently delivered to its destination. For example, in Chord, the node with $nodeId$ maintains a *routing* or *finger table* that contains the $O(\log n)$ tuples of the form $entry_i = (nodeId_i, IP_i)$, where $nodeId_i = root(nodeId + 2^{i-1})$. In addition, each node with ID x maintains pointers to its immediate predecessors, denoted in order of distance $p^1(x)$, $p^2(x), \ldots, p^i(x)$, and a list of its nearest successors in the Id space, denoted by $s^1(x), s^2(x), \ldots, s^i(x)$. We follow Chord in using this *constrained* routing table; for a discussion of the performance implications of this decision, and possible optimizations, see section 6.4.

To route a message to $root(k)$, Chord finds the "finger" in its routing table with the highest $nodeId$ less than or equal to k and hands over the query message to that node to be routed further. At the end, the destination (supposedly $root(k)$) replies to the sender via direct IP communication. During this routing, the distance between a message's current location and its destination is halved in each hop resulting in a logarithmic number of hops. In this approach, called *recursive routing*, the sender delegates routing to the next hop and from then on it loses control over the traversed hops. Instead of asking to forward the message, the sender may ask for the information regarding the next hop. In this approach, called *iterative routing*, the sender discovers the full route to $root(k)$ and contacts the destination.

2.2 Security Issues in DHTs

Like other networks, DHTs are vulnerable to attacks. Below, we briefly overview the attacks (specific to DHTs) proposed in previous work [23, 24] and the known approaches to dealing with them.

Sybil attack [25]: an attacker generates a large number of bogus DHT nodes to out-number the honest nodes. This attack is, in general, overcome by introducing an *off-line* trusted entity [23], such as a certificate authority (CA). Even with a CA, if malicious nodes can pick their nodeIds, they can control the access to popular data objects by becoming the root of those objects' keys. Thus it is typically assumed that the CA will perform some level of admission control to limit the number of certificates issued to attackers.

Message corruption, drop, and delay [23]: A DHT node forwards messages (data as well as control) for others using its routing table. An attacker can eavesdrop on and modify overlay messages passing through it. Even if the messages are signed and encrypted, he can drop or delay them. Iterative routing can be used to prevent such attacks on routing messages [2, 26].

Routing Table Poisoning (Eclipse Attack) [23]: Since a node's routing table is generated from information from other nodes, it is possible that its routing table could be corrupted (i.e. filled with attacker's IP addresses). This attack is effective for DHTs with flexible routing tables.

Root Spoofing: Routing in a DHT is "proximity" routing. A message is routed to a key's root rather than a node specified by the querier. Without detailed knowledge of the replier's neighborhood, the querier cannot verify whether the replier is indeed the root of the key.

3. PROBLEM DESCRIPTION, ASSUMPTIONS AND OVERVIEW

DHTs are designed in such a way that each node stores information about only a small number of other nodes. This makes them scalable in terms of storage overhead and routing overhead, but leaves nodes vulnerable to attacks based on limited knowledge of the current state of the network, such as root-spoofing. Here we outline a specific (known) attack scenario, followed by a general definition of routing security and a description of our solution and assumptions.

3.1 Root Spoofing Attack

The most important property in DHT routing is that when a lookup for a key k is performed, the resulting DHT node should be $root(k)$, given the *current* system state. Figure 1 shows a typical attack scenario, where node R is $root(k)$. Suppose C inserts a key-value pair (k, v) at R. Later, when Q queries the key k, it asks D for the next hop, which returns E. Now E, colluding with B (or perhaps unaware of R's existence), returns B, which claims (being close to k) that it is responsible for k, thus, effectively hiding R from Q. Since Q does not know about R and B appears to be a plausible destination, Q accepts B as the destination responsible for that key. In this case, Q is unable to retrieve the value v corresponding to key k. We note that this attack can also work if any node along the query route, for example D, is malicious: while D cannot claim to be $root(k)$ he can route the query to his colluder B who is close to k.

Thus a DHT without a root verification mechanism cannot guarantee the delivery of query messages. The same basic attack can also be applied to insertion of (key, value) pair, routing table maintenance and membership events, because typically all of the DHT protocols rely on the ability to find the correct root of a key. Thus, without a root verification mechanism, query and insertion messages could be delivered to incorrect nodes, routing tables could have more and more malicious nodes, and a new node could join a logically separate network filled with malicious nodes. In short, an adversary can use this attack to disrupt most of the functionalities of the DHT.

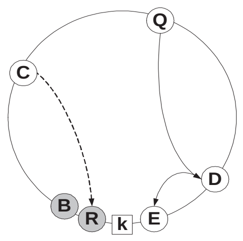


Figure 1: Root Spoofing Attack

3.2 Problem Description

Security of DHT routing is thus a weak link in building secure DHT-based applications. Castro *et. al.* [23] define routing security as follows: *The secure routing primitive ensures that when a non-faulty node sends a message to a key k, the message reaches all non-faulty members in the set of replica roots R_k with very high probability.* While this is certainly a necessary condition for security, we argue that it does not specify sufficient conditions. For example, if we were to design a "secure routing primitive" that guaranteed delivery to all replica roots in $\Omega(n!)$ steps, it would meet this definition but its performance would be completely unacceptable, essentially making the protocol one giant algorithmic denial-of-service attack [27]. Thus efficiency in the face of an attack is an important concern. With this in mind, we refine this definition more formally as follows:

DEFINITION 1 (δ-SECURE DHT ROUTING). A routing protocol is δ-secure if it ensures that with probability at least $1 - \delta$, when a non-faulty node A initiates a lookup for a uniformly chosen key k, A correctly identifies the node $root(k)$ within an expected $O(\log n)$ hops, despite the presence of a fraction $f < 1$ of faulty nodes[1].

Notice that the above definition explicitly states that we will *allow* a certain fraction, δ, of queries to fail. Thus an important question is what values of δ a protocol can support. Later in this paper, we show that Myrmic can deliver 99.99% of packets within 1.4 times the latency of Chord even with 30% of attackers dropping packets. Also note that we allow arbitrary behavior by malicious nodes, including root spoofing; message corruption, dropping, or mis-routing; and other behavior outside of the protocol specification. In summary, (1) the sender must be able to verify the root, (2) in case root verification fails (e.g., a malicious node impersonates the root), the message must be able to bypass malicious nodes and eventually reach the root if it is *reachable* (that is, there is a path from the sender to the root that consists of only non-faulty nodes), and (3) the secure routing protocol must be efficient, even under attack. We note that an important technical consequence of our definition is that protocols satisfying this definition will correctly identify the root, even when the root node is malicious.

3.3 Assumptions and Attack Model

To avoid attacks related with nodeId assignment such as the Sybil attack [25], we assume the existence of an off-line certification authority as in [23]. We also assume that N nodes form a DHT network. A bounded fraction of the nodes f $(0 \leq f < 1)$ may be faulty. We assume faulty nodes may collude and adversaries are "non-adaptive", i.e., (1) faulty nodes can operate in concert. (2) at most a fraction f of current nodes are malicious or vulnerable to compromise in any given time period, including the initial time period when the network bootstraps. The set of vulnerable nodes, however, are not chosen by the adversary. We note that a "fully adaptive adversary" that can instantaneously corrupt any node in the DHT can defeat the security properties of not only Myrmic, but all previous protocols in the literature. We assume adversaries can not corrupt or prevent IP network layer communication between honest DHT nodes. However, we allow adversaries complete control of IP- and DHT-layer traffic passing through faulty nodes. (In other words, they can drop, delay or replay messages, attempt to route messages to their collaborators, *etc.*) We assume that all nodes are loosely time synchronized, e.g. honest nodes' clocks agree to within a few seconds.[2] We do not consider denial-of-service attacks (against arbitrary nodes) at the network level; these attacks can essentially defeat any protocol in the literature, (e.g. by preventing nodes from initiating lookups) and are outside the scope of this paper.

3.4 Myrmic: High-Level Overview

In addition to the off-line CA, Myrmic introduces a new on-line authority, called the *Neighborhood Authority (NA)*.

[1]Node faults may be arbitrary, including directed adversarial behavior as well as faults caused by environmental conditions such as node failures or transient routing failures.

[2]Since our setting includes a small set of trusted nodes that communicate periodically with each host, it is feasible to provide this level of synchronization

The NA only participates in DHT network management by issuing *Neighborhood Certificates (nCerts)* to small sets of nodes after DHT membership events such as joins and leaves. The NA is *not* involved in any other functionality of DHT routing, and in particular queries proceed without contacting the NA. The NA has a public/private key pair for signing certificates and we assume that its certificate is publicly available. The NA is stateless, so that it can easily be replicated to handle high churn rates or transient node failures. Similar to a Certificate Authority, the NA is a central point of trust rather than a central point of failure.[3] If the NA goes offline for some period of time, there will be two effects. First, new nodes will be unable to join the network (but can still route queries through existing network nodes). Second, since our analysis treats nodes that leave the network without communicating with the NA as faulty, long periods of unavailability will increase the fraction of faulty nodes the network must tolerate. Once a node is no longer included in fresh nCerts, it is no longer considered faulty, because no nodes will attempt to contact it.

Myrmic uses *iterative routing*, which incurs just under twice the latency and message cost of recursive routing, in order to allow a querier to monitor query progress and to find alternative routes in case its query is mis-routed or dropped on the route. With iterative routing, the secure routing problem can be reduced to the problem of verifying that a query for key k makes progress and discovers the correct $root(k)$. Myrmic allows a querier to verify the keys a node is currently responsible for using an nCert issued by the NA. This prevents a malicious node from impersonating the root of a key outside it is not responsible for or routing a query to an incorrect node.

A naïve approach would be for the NA to issue a nCert to a joining node specifying the range of the keys it will be responsible for. Whenever the node needs to prove that it is responsible for a certain key, it could present its nCert. However, in such an approach it is not clear how to deal with certificate revocation securely and efficiently: when a new node B joins the network, it is assigned a part of the key range that was previously assigned to another node A, which necessitates revocation of part of A's previous nCert. Without efficient and secure revocation of nCerts, a malicious node may claim responsibility for a key by presenting an old certificate. If the revocation information is broadcasted by trusted nodes, nodes will have to keep an amount of information linear in the number of revoked certificates. Furthermore, the NA will be required to remember what nCerts it has previously issued in order to revoke them, increasing the complexity of implementation and NA replication.

Therefore, the key problem here is how to allow queriers to efficiently obtain a fresh certificate that explains the current range. Myrmic enables queriers to find fresh information by checking with "authorized witnesses." We choose a node's neighbors as the authorized witnesses because: (1) a node's range is determined by its neighborhood information, (2) the root's neighbors are often used as replica roots, and (3) nodes are already required to maintain neighborhood information. The IP addresses of these neighbors are listed in the root's nCert, which is a signed list including the root and its immediate neighbors. When a certificate is invalidated by a change in membership, those neighbors are informed. Hence as long as a malicious node has one honest neighbor, it cannot use a revoked certificate since the querier can contact any neighbor directly to provide a more recent certificate; thus by adjusting the neighborhood size appropriately, we can limit the probability that a malicious node can use a revoked certificate while not requiring any interaction with the NA. We stress that Myrmic also includes protocols that allow the DHT to quickly recover from the occasional event that all the nodes in a neighborhood become faulty; see Section 5.4.

4. SECURE DHT ROUTING

In this section, we first present the format of the nCerts used by Myrmic to certify the range of a node. We then present the algorithms employed for the root verification, join, leave, and lookup operations.

4.1 Root Verification Using nCerts

The range of a node in Myrmic is determined by its nodeId and that of its immediate neighbors: the range of node R is the interval from the predecessor of R ($p(R)$) to R, i.e., $range(R) = (p(R), R]$. Thus, in a DHT with static membership, R can prove its range by presenting a signed nCert that includes the nodeIds of R and $p(R)$. With dynamic membership, however, a node's range may change, requiring a method for queriers to determine whether an nCert is fresh.

For this purpose, we include several nodes in each nCert to serve as witnesses to the freshness of the nCert. When a nCert is revoked, the witnesses are notified by the NA. Hence, by consulting with the witnesses, one can verify that an nCert is fresh. Since it only takes a single witness to prove that an nCert has been revoked, a malicious node can only use a revoked nCert if all of its witnesses are faulty. Hence by adjusting the number of witnesses, we can bound the probability that a malicious node successfully uses a revoked nCert. As previously mentioned, we choose a node's nearest neighbors as its witnesses because they must maintain this information anyways and may also be replica roots. The nCert format is thus:

$$\begin{aligned} nCert_R &= Sign_{sk_{na}}\{nList_R, issueTime, expireTime, pk_R\} \\ nList_R &= \{I_{p^l(R)}, \ldots, I_{p(R)}, I_R, I_{s(R)}, \ldots, I_{s^l(R)}\} \\ I_R &= (nodeId_R, IP_R) \end{aligned}$$

An nCert is signed by the NA using a secure digital signature $Sign_{sk_{na}}(\cdot)$ with the private key of the NA. The nCert also includes its issue time, its expiration time, and the public key of R The $nList_R$ in $nCert_R$ includes tuples $I = (nodeId_i, pk_i, IP_i)$, which allow direct IP connections

[3]If there is no appropriate central point of trust, the stateless design of the NA also simplifies a threshold cryptography-based implementation; however, this is outside the scope of the present paper

```
     // verify if $R$ is the root of key $k$
     $Q$.verify_root($nCert_R$, $k$)
 1:      if($good\_timestamp\_signature(nCert_R)\ is\ false$ or
 2:          $is\_root(nCert_R, k)\ is\ false$)
 3:          return $false$;
 4:      for($X \in nCert_R.nList$)
 5:          $nCert'_R = X.find\_nCert(R)$;
 6:          if($good\_timestamp\_signature(nCert'_R)\ is\ true$)
 7:              if( $nCert'_R.issueTime > nCert_R.issueTime$
 8:                  and $is\_root(nCert'_R, k)\ is\ false$)
 9:                  return $false$;
10:      return $true$;

     // verify if $R$ is the root of $k$ according to $nCert_R$
     $Q$.is_root($nCert_R$, $k$)
11:      if($k \in (nodeId_{p(R)}, nodeId_R]$)
12:          return $true$;
         return $false$;
```

Figure 2: The pseudocode to verify if R is the root of k.

to R's neighbors. The size of the $nList$ $(= 2l + 1)$ is a system parameter defined by the NA. The relation between this parameter and security is described in Section 5.

Figure 2 summarizes the root verification procedure, which assumes that all nodes have current nCerts. In this procedure, querier Q uses $nCert_R$ to verify whether $R = root(k)$ using two tests: first, it checks if $k \in range(R) = (p(R), R]$ where $p(R)$ and R are included in $nCert_R$; second, it checks if $nCert_R$ is fresh. These two tests are accomplished by obtaining copies of $nCert_R$ directly from the witnesses. If a witness X gives a more recent, valid $nCert'_R$ that lists a different $root(k)$, then R fails the test (lines 7-9). The querier Q's communication overhead is to contact $2l$ witnesses and the computation overhead is at most $2l + 1$ signature verification.

4.2 Neighborhood Certificate Update

For every membership change, the NA must re-issue nCerts to the nodes affected by the change. When a node J joins the DHT, it obtains a nCert and, in addition, the NA must update the nCerts of the $2l$ nodes that gain J as a neighbor. More specifically, the NA updates the nCerts of all the nodes listed in $nList_J$ to include J in their $nList$s. Similar updates are also carried out when the NA is notified that J has left the network.

In Chord, when a new node joins, it first learns its neighborhood information from a bootstrap node, and then gradually fills in its finger table using queries to the appropriate Ids. Myrmic's join protocol modifies only the first part of the Chord joining protocol, i.e., we only modify the part of the protocol that the joining node follows to initiate the list of its neighbors and notify them.

As shown in Figure 3, with the help of a bootstrapping node B (not necessarily trusted), the joining node J locates the node $R = root(nodeId_J)$ using the secure iterative routing protocol (to be presented in section 4.3). Next J contacts the NA to get $nCert_J$.

To generate $nCert_J$, the NA needs to learn the $(nodeId, IP)$ pairs of the $2l$ nearest neighbors, which will be in J's $nList$. Similarly, to update the nearest neighbors' nCerts, NA needs to find out the $(nodeId, IP)$ pairs of *their* nearest neighbors (line 6). For this purpose, NA obtains and verifies R's nCert using the root verification protocol (line 5). Once NA has

```
     // node J joins the network. node B is used for bootstrap
     $J$.join($B$)
 1:      $R = B.find\_root(J)$;
 2:      $NA.update\_nCerts(R, J)$;
 3:      $init\_finger\_table(B)$;
 4:      $update\_others()$;

     // issue a nCert to the joining node and update its neighbors' nCerts
     $NA$.update_nCerts($R$, $J$)
 5:      if ($accept(J)$ is $true$ and $verify\_root(nCert_R, J)$ is $true$)
 6:          $list = construct\_neighbor\_list(R, J)$;
 7:          $generate\_distribute\_nCerts(list)$;

     // $NA$ constructs a list of the nearest neighbors of the joining node
     $NA$.construct_neighbor_list($R$, $J$)
 8:      $list = nil$;
 9:      for ($X \in R.nCert$)
10:          for ($Y \in X.nCert$)
11:              for ($Z \in Y.nCert$)
12:                  if ($in\_list(list, Z)$ is $false$) and $ping(Z)$ is $live$)
13:                      $insert\_into\_list(list, Z)$;
14:      $insert\_into\_list(list, J)$;
15:      while($count\_live\_successors(list, J) < 2l$)
16:          $get\_more\_successors(list, J)$;
17:      while($count\_live\_predecessors(list, J) < 2l$)
18:          $get\_more\_predecessors(list, J)$;
19:      return $list$;

     $NA$.generate_distribute_nCerts($list$, $J$)
20:      $nList_J = gen\_nList(list, J)$;
21:      for ($X \in nList_J$);
22:          $nList_X = gen\_nList(list, X)$;
23:          $nCert_X = Sign_{sk_{na}}\{nList_X, issueTime, expireTime\}$;
24:          send($nCert_X$);

     // these procedures are not modified from chord
     $J$.init_finger_table($B$)
     $J$.update_others()
```

Figure 3: The pseudocode for a node J joining the DHT.

```
     // node X maintains updated neighborhood
     $X$.maintain()
 1:      for ($Y \in nList_X$)
 2:          if ($ping(Y)$ is $dead$)
 3:              $Z = Y$;
 4:              do
 5:                  $Z = find\_root(X, (Z.id + 1))$;
 6:              while ($ping(Z)$ is $dead$)
                 $NA.update\_nCerts(Z, Z)$;
```

Figure 4: The pseudocode for a node X to maintain updated neighborhood

R's nCert, it can directly contact the neighbors of R. Next NA calls $construct\_neighbor\_list()$ to construct a list of nodes including J and its (at least) nearest $4l$ neighbors ($2l$ predecessors and $2l$ successors). The result is a list: $list = \{\ldots, I_{p^{2l}(J)}, \ldots, I_{p(J)}, I_J, I_{s(J)}, \ldots, I_{s^{2l}(J)}, \ldots\}$. This list includes all the information the NA needs to generate new nCerts. Once NA has the list, it calls $generate\_distribute\_nCerts()$ to generate the $2l + 1$ new nCerts for J, its nearest l predecessors, and its nearest l successors. Each $nCert_X$ is sent to all of the nodes in its $nList$.

When a node *leaves* the DHT, other nodes (gradually) update their state tables. The range of the absent node must be allocated to its neighbor(s). Hence once a leave is detected, the NA should be notified to update the nCerts of the neighbors of the missing node. In Myrmic, a node X periodically calls $maintain()$ to ping the nodes listed in its nCert (Figure 4). If it believes that one of them, say Y, has left, it finds Y's immediate live successor Z and contacts the NA, which calls $update\_nCerts(Z, Z)$. When the procedure finishes, Z inherits $range(Y)$ and the nodes listed in $nCert_Z$ (including X) obtain updated nCerts.

We briefly consider the overhead of $update\_nCerts$. To

```
        // find the root of k using gateway node G
        $Q$.**find_root**($G, k$)
1:          $try = G;$
2:          $nCert_{next} = try.next\_hop(k);$
3:          $nCert_{current} = nCert_{next};$
4:          do
5:              if($good\_timestamp\_signature(nCert_{next})\ is\ true$)
6:                  if($is\_root(nCert_{next}, k)\ is\ true$)
7:                      if($verify\_root(next, k)\ is\ true$)
8:                          return $next$;
9:                      else
10:                         $nCert_{current} = nCert_{next};$
11:                 else
12:                     if($progress(try, nCert_{next}, k)\ is\ true$)
13:                         $nCert_{current} = nCert_{next};$
14:             $try = random\_select\_from(nCert_{current});$
15:             $nCert_{next} = try.next\_hop(k)$
16:             $mark\_as\_contacted(nCert_{current}, try);$
17:         while($try\ is\ not\ nil$);
18:         return $nil$;

        // return root or closest preceding finger
        $X$.**next_hop**($k$)
19:         for($nCert_i \in \{nCert\ stored\ by\ X\}$)
20:             if($is\_root(nCert_i, k)\ is\ true$)
21:                 return $root$;
22:         return $closest\_preceding\_finger(k)$

        // check if $try$ returns the closest proceding finger to $k$
        **progress**($try, nCert_{next}, k$)
23:         $i = \lfloor log(next - try) \rfloor;$
24:         if($2^i < k - try < 2^{i+1}$ and $is\_root(nCert_{next}, try + 2^i)\ is\ true$)
25:             return $true$;
26:         return $false$;

        // this procedure is not modified from chord
        **closest_preceding_finger**($k$)
```

Figure 5: The pseudocode to find and verify the root.

simplify the discussion, we calculate communication and computation overhead in terms of the number of messages an entity sends and the number of digital signature operations it performs, respectively. Note that nodes in neighbors' nCerts are overlapping and $nCert_X$ is stored by all nodes listed in it. Hence, in the three for loops of $update\_nCerts()$, the NA can retrieve all of the $4l + 1$ nCerts held by the $2l + 1$ nodes listed in $nCert_R$. The NA pings another $2l$ nodes in $count\_live\_predecessors(list, J)$ and $count\_live\_successors(list, J)$. The NA also signs the $2l + 1$ newly generated nCerts. Since an nCert is sent to all nodes listed in it, the NA sends $4l + 1$ messages to distribute the $2l + 1$ nCerts. The overhead of regular nodes involved in the change is (at most) one signature verification and one message, either sending an nCert to NA or replying to a ping.

4.3 Secure Iterative Routing

Using the root verification procedure along with securely maintained finger tables, we can design an iterative routing mechanism that guarantees correct and efficient DHT lookup: at each hop, the Querier can verify, via nCert validity, that the current node provides a correct next hop, and at the conclusion, the querier can use root verification to check that the alleged result of a query for k is the correct $root(k)$. Figure 5 shows a procedure for a Myrmic client Q to route a query for key k through a possibly untrusted gateway G (when Q's routing table is not established) or Q itself.

The protocol resembles the iterative routing of Chord. The main idea is as follows: Q iteratively asks a node on the route for the nCert of the next hop (using the `X.next_hop` procedure), which is either the root or the closest preceding finger towards the key. For each $nCert_{next}$ received, Q first verifies, locally, if $nCert_{next}$ shows $next$ as the root of the key k (line 6). If so, Q verifies the freshness of $nCert_{next}$ using root verification (line 7). Otherwise, Q checks if $nCert_{next}$ makes the expected progress, i.e., if try indeed returned its closest preceding finger to k (line 12). If so, Q randomly selects a node from $nCert_{next}$ for its next hop (line 14). Otherwise, Q discards $nCert_{next}$ and randomly selects a node from the previous $nCert_{next}$ (i.e., $nCert_{current}$). Note that, while verifying the progress, Q does not need to verify the freshness of $nCert_{next}$, because as long as one of the nodes listed in $nCert_{next}$ is honest and alive, Q will be able to find a next hop. Hence Q only verifies the signature and timestamp of $nCert_{next}$ (line 5).

In addition to the protocol shown in figure 5, our iterative routing protocol also uses dual timeouts. A *soft timeout* happens when Q judges that it has waited too long for a reply from *try*, and simply contacts another node listed in $nCert_{next}$. If Q receives a message from try after a soft timeout, it will still process the result. When all nodes in $nCert_{next}$ have been marked as contacted and the *hard timeout* is reached, Q determines that the lookup has failed and drops the query. Using a short soft timeout may increase the number of $next\_hop$ (NH) messages sent, while reducing the delay caused by a slow link or, more importantly, a malicious node who does not respond in time. On the other hand, by using a longer soft timeout, we can reduce the total number of NH messages sent; this gives the node Q the opportunity to trade off lookup latency for bandwidth consumption. (See appendix A for more discussion).

5. SECURITY ANALYSIS

In this section, we analyze the security of the protocols sketched in section 4. We first show that honest nodes always have a correct nCert and consistent neighborhood view. This allows us to prove that the root verification procedure fails with only small probability. Finally we argue that because of these properties, the iterative routing procedure of Myrmic succeeds in $O(\log n)$ steps with high probability.

5.1 Security of nCert Updates

We define a correct nCert to be one that consists of the nodeID and IP of its owner, plus the l most immediate predecessors and successors that are visible to the NA. We argue that with high probability the Neighborhood Certificate Update protocol generates correct nCerts. In this protocol, the NA first constructs a neighbor list and then generates and distributes nCerts based on the list. The second step is straightforward assuming that adversaries cannot corrupt or prevent the delivery of IP-network layer communication between honest DHT nodes and the NA. Hence the correctness of this protocol depends only on the NA constructing a correct neighbor list.

So suppose that at time t, all honest nodes possess correct nCerts, and that at time $t + 1$, honest node n notices that its predecessor does not respond to pings and initiates the nCert update protocol by sending its nCert plus the nCert of all of its neighbors. The NA responds by contacting all of these neighbors and asking for their nCerts, and pinging all nodes mentioned in the nCerts received in these steps. Altogether, $4l$ nodes will be contacted, $2l$ of which should have nCerts listing each node in the neighborhood of n. Thus any node in the local neighborhood can only be obscured if $2l$ consecutive nodes are faulty or the nCert signature scheme admits forgeries. By assumption, the probability of the latter is negligible; by our model, the probability of the former, when fraction f of nodes are faulty,[4] is at most f^{2l}. Thus with high probability the NA discovers all neighbors of n and issues a correct nCert to each affected node at time $t+1$. A similar argument establishes the correctness of nCerts after joins.

5.2 Security of Root Verification

Let us assume that nCerts are unforgeable, and consider the circumstances under which a node R may falsely claim responsibility for a key. Since nCerts are unforgeable, and the NA is trusted, R may only fraudulently claim or disclaim responsibility after a change in membership. Node R's range may change in one of four ways. (1) A new node J joins the DHT and becomes R's predecessor. R loses part of its previous range. (2) R's predecessor left and R's new range includes both its old range and its old predecessor's range. (3) R left and lost all its range. In these three cases, NA runs the nCert update protocol and new nCerts are distributed to all witnesses. To use a revoked nCert, a malicious node must collude with all "current" witnesses. Let T be the lifetime of a nCert, and let f include the percentage of nodes leaving during T. Assuming malicious nodes do not leave, the probability that a revoked nCert can be used is f^{2l}, i.e. the probability that all $2l$ neighbors of R are faulty. (4) R is relocated. A node may be relocated only when its nCert has expired and it is trying to obtain a new one. In this case, R cannot claim responsibility for its previous range because of the expired nCert.

5.3 Security of Iterative Routing

Let δ be a "security parameter" for secure DHT routing, e.g. the probability of routing failure we are willing to tolerate.[5] Here we show how to set the Myrmic parameter l (as a function of n and f) so that with probability at least $1-\delta$, the expected number of steps for any query is $\frac{1}{2(1-f)}\log n$.

First, we note that when Q contacts the node try in the iterative routing step, Q requests node try's closest preceding finger to k *and* the nCert of the finger. Thus Q can check if the finger returned by try is in fact the node responsible for the key $nodeId(try)+2^i$ where $2^i < k - try < 2^{i+1}$. If it is not, the node try can be regarded as faulty and ignored. Thus without loss of generality we can treat faulty nodes as "black holes" that do not respond to queries, when considering the query protocol.

Now, we define the *chord next hop* from node n to key k to be the next hop of n that (iterative) Chord would query in a fault-free environment. Notice that the *chord path* of chord next hops always has length at most $\log n$ and has expected length $\frac{1}{2}\log n$. We say that a Myrmic lookup *follows the chord path* if at each step it contacts a node in the neighborhood of the chord next hop. A lookup that follows the chord path will also take on average $\frac{1}{2}\log n$ "hops" (walking randomly about the chord hops) but may spend multiple steps discovering the correct next hop. We will prove that a Myrmic lookup follows the chord path with probability at least $1-\delta$ and spends on average $O(1)$ steps at each hop, completing the proof.

First, we note that Myrmic only fails to follow the chord path when some chord next hop and all $2l$ of its neighbors are faulty. Since there are at most $\log n$ chord next hops, and each has a faulty neighborhood with probability at most f^{2l+1}, by the union bound the probability of such failure is at most $f^{2l+1}\log n$. Thus setting the neighborhood size

$$2l+1 = \left\lceil \frac{1}{\log(1/f)}\left(\log\log n + \log\frac{1}{\delta}\right)\right\rceil \qquad (1)$$

will give the desired probability of following the chord path. Given that a Myrmic lookup follows the chord path, the expected number of nodes contacted at each hop is $\sum_{i=1}^{2l+1} i\cdot(1-f)f^{i-1} \leq (1-f)\sum_{i=1}^{\infty} i\cdot f^{i-1} = \frac{1}{1-f}$. This gives the desired bound.

We note that in case the Myrmic lookup does not follow the chord path, it may still successfully complete in a short time; thus this analysis *understates* the success probability when l is set appropriately.

5.4 Bad State Recovery

Our definition of secure routing explicitly allows some queries to fail due to a neighborhood consisting entirely of faulty nodes; in particular, we expect that roughly $f^{2l+1} = \delta/\log n$ fraction of neighborhoods will be "corrupt" in this manner. One method of dealing with this would be to set $\delta = n^{-\log n}$, so that the probability of having any corrupt neighborhoods is negligible; this would significantly increase the cost of Myrmic routing. Instead, we deal with this situation by periodically relocating each node to a different part of the ring, so that with high probability a neighborhood that is corrupted in one time period will not be corrupted in the next period. This "induced churning" [28, 29, 30] allows Myrmic to tolerate some corrupted neighborhoods by ensuring that these failures will be transient.

In order to implement this scheme, two mechanisms are needed. The first is a way to determine when and to where a node should be relocated. Whenever a node's nCert expires, it must contact the NA and have a new certificate issued. The NA may then decide to relocate the node based on some verifiable but unpredictable information; an example is an NA signature on the beginning time of the current

[4] for purposes of correctness, we treat faulty nodes and adversarial nodes identically, i.e., a node that goes offline during the duration of an nCert with probability f is considered to be faulty.

[5] Here we define a routing failure as the event that after some fixed number of steps $f(n) = \Omega(\log n)$ a lookup query has failed to identify the correct mapping between a key and a node.

period. If the hash of this signature and the node's certificate exceeds the fraction of time elapsed in the period, the node is assigned a new nodeID, by hashing the node's certificate and IP address with the unpredictable information. Thus anyone can verify, given the NA signature, that a node should be relocated and what its new nodeID should be.

The second mechanism that is needed is a protocol for recovery when a node joins (or is relocated to) a corrupt neighborhood. If the malicious node R's neighborhood is corrupted, it can effectively prevent a new node J from joining its range until its current $nCert_R$ expires, since no one in its neighborhood will contradict the revoked $nCert_R$. Once $nCert_R$ expires, it will have to be renewed, and the NA will contact the $2l$ nodes on each side of R; if one of these is honest, the new $nCert'_R$ will include J. However, there is a small probability f^{4l+1} that all $4l$ neighbors of R are corrupted, and in this case R will continue to be able to prevent J from joining its range until it is relocated (at which time it will no longer be issued an nCert for the range covering J). Until all of the $2l$ corrupted nodes surrounding R are relocated, corrupt nodes will continue to cover J (because of the statelessness of the NA). Once all nodes have left the neighborhood, new nodes will be unable to find a valid nCert for the range. In this case, a recovery protocol can be invoked: a node joining with ID J conducts a binary search to the left and right of J for the nearest valid nCerts of a predecessor P and successor S of J. Once it obtains these, it contacts the Neighborhood Authority with the nCerts and finger tables of S and P, and the Neighborhood Authority builds a neighborhood list that extends $2l$ nodes before P and $2l$ nodes after S before issuing $nCert_J$. The expected number of hosts contacted is $4l + 2/(1 - f)$, and the NA should refuse to repair a neighborhood of size larger than l^2 (the probability of a neighborhood of this size being compromised is negligible). Finally, in order to allow other nodes to find the fresh $nCert_J$, J uses lookups to identify the $O(\log n)$ nodes that should have J as a finger and sends them $nCert_J$.

Thus, with probability at least $1 - f^{4l+1}$, a corrupt neighborhood can be repaired in at most two time periods; the expected load on the NA from the repair protocol is less than the cost of a node leaving and then joining again, and never more than l^2.

6. IMPLEMENTATION AND EVALUATION

Our implementation of Myrmic consists of two indenpendent components: the DHT node and the NA. Both components are implemented in C and use Openssl 0.9.8 [31] for RSA digital signatures with the SHA-1 hash function. Myrmic's DHT node component is based on the i3-Chord implementation [32]. We evaluated the performance of our prototype implementation 1) on PlanetLab [21] to evaluate its performance on a wide-area network and 2) on a local testbed to assess the robustness of Myrmic under attack.

6.1 Parameter Selection

To select timeout values, we measured the pairwise ping time of PlanetLab nodes. Each node sent 10 pings to every node in our experimental setup. The average, median, 75th- and 99th-percentile ping times were 54, 50, 78, and 177 ms. Based on these results, we chose 2000 ms for the hard timeout value, 200 ms for the nCert verification timeout, and 78 ms for the soft timeout. The long hard timeout and nCert verification timeouts ensure that the querier does not miss replies from intermediate hops and neighbors of the root, while the shorter soft timeout prevents wasted time in iterative queries, as explained in section 4.3 and appendix A.

To select T_{ncl}, the life time of a nCert, we studied the relationship between T_{ncl}, median node session times, and nCert size. The session time of a node is the difference between the time when it joins the network and the time when it leaves the network. Median node session times of a typical file-sharing P2P application can be found in published studies (e.g. [33]). We assume that node joins and leaves both follow a Poisson process, with the same event rate λ, resulting in a stable network size. Under this model, the event rate for median session time T_{ms} is

$$\lambda = \frac{n \times \ln 2}{T_{ms}} \tag{2}$$

Hence the fraction of nodes that join or leave the DHT during a period of length T_{ncl} is

$$\alpha = \frac{n \times \ln 2}{T_{ms}} \times \frac{T_{ncl}}{n} = \frac{\ln 2 \times T_{ncl}}{T_{ms}}. \tag{3}$$

Given f, α, n, and δ, we can compute nCert size using equation 1. For example, if $f = 10\%, n = 5000, \delta = 0.005$, then we obtain the nCert size $2l + 1 = 7$ by setting $\frac{T_{ncl}}{T_{ms}} = 0.1$. Note that smaller values of $\frac{T_{ncl}}{T_{ms}}$ require more frequent renewal of nCerts. Appendix B discusses this issue in detail.

6.2 Wide-area Evaluation

Expermental setup. Each of our wide-area experiments was run using a set of approximately 120 PlanetLab machines[6] as Myrmic nodes, without using the Sirius calendar service [21], and a single machine in our local testbed as the NA, running Ubuntu Linux (2.6 kernel), with a 3GHz Pentium IV CPU and 1GB RAM. Due to space limitations, we only show the results of experiments using the single set of parameters selected in section 6.1. In each of the experiments, we first join every node to the network using the NA, and then a simple test program built on top of a DHT node is used to periodically send random queries. Each node sends 500 queries, one query every 3 seconds, making the total number of query messages about 60,000.

Overall Performance Figure 6 (a) shows the *query response time*, defined as $(t_f - t_r)$ where t_r is the time when the DHT layer of node Q receives the query request from the test program and t_f is the time when the DHT node reports the query result to the test program after completing the nCert verification. The 97th and 90th percentiles are 346ms and

[6]All of these nodes were located in North America, to ensure relatively uniform delay, which simplifies the analysis of the prototype implementation.

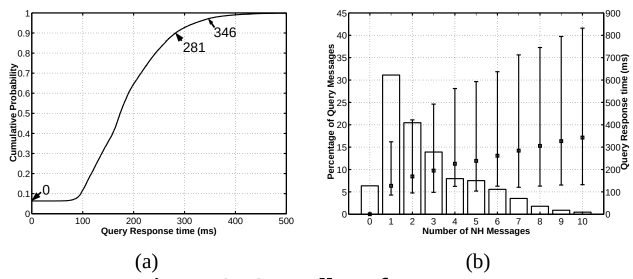


Figure 6: Overall performance

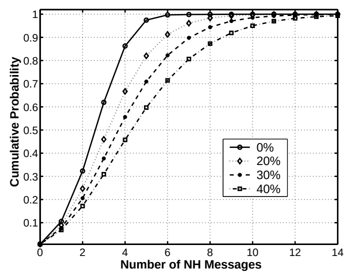
281ms. About 6$\%$ of the queries are finished immediately. This happens when $root(k)$ is the querier node Q or one of its neighbors in $nCert_Q$. For these cases, the correctness of the $root(k)$ was verified at the time that Q received $nCert_{root(k)}$. In a network with N nodes, each node can handle $\frac{w}{N}$ fraction of queries locally.

For the rest of the queries, Q sends one or more NH requests. We categorize the queries based on the number of NH requests and show both the response time of queries and percentage of queries in each category. Compared to Chord, which approximately follows a normal distribution for the number of hops, Myrmic has a distribution shifted to the left. As shown in figure 6 (b), 93$\%$ of messages are delivered within 6 NH requests (not including the root verification message). Figure 6 (b) also shows the min, median, and 97th percentile of query response time in each category. As expected, the medians are approximately linear in the number of NH requests.

Evaluation of the NA. To determine the capacity of our NA, we ran a set of experiments with increasing churn rates. For each experiment, we recorded the average join time, and we determined the capacity of the NA by finding the churn rate that caused this join time to increase significantly. In each of these experiments, we start with 1000 DHT nodes on 100 planetlab hosts. Then on each machine, a node is killed periodically and a new node joins the network immediately, causing both a leave and a join event. The results of this evaluation suggest that our NA (running Ubuntu Linux with a 3GHz Pentium IV CPU and 1GB RAM) can handle as many as 34 events (17 joins and 17 leaves) per second, for a "Churn rate" of 17 [33]. Since the cryptographic computations required for an nCert update – 7 signatures and 14 verifications – take about 25.3 ms on the testbed NA, 34 events require around 860 ms of computation, which supports this conclusion. Plugging the median session times reported in [33] (1 minute to 1 hour) into Eq. 2, we find that one NA can handle a total number of nodes ranging from 1472 to 88299. Note that these experiments were performed using an ordinary desktop machine rather than a high-performance server, and our threaded implementation has not been optimized for performance. Thus we expect that on multicore or multiprocessor systems an optimized NA should be capable of handling a significantly higher workload. Finally, we note in Appendix C that it is possible to replicate the NA to further increase scalability.

6.3 Local Network Evaluation

One of the most salient features of Myrmic routing is its efficiency and robustness even with a significant fraction of adversaries. To validate this claim, we ran a series of experiments, in a local lab with 34 PCs, with a 1000 node DHT and varying fractions of "black hole" adversaries that dropped all NH requests. Each node sends 1,000 queries resulting in 1,000,000 queries for each experiment. The results of the experiment are shown in Figure 7. The graph shows the cumulative distribution of the number of NH requests for each experiment (when the size of nCerts is 7), while the table shows the percentage of failed queries for nCerts of size 7 and 11, compared with the failure bounds (in parentheses) computed in Section 5.3. As expected, the failure rates and NH counts are both improvements on our worst-case bounds. For example, with $w = 7$, even when 40% of nodes drop all requests, less than 1% of queries fail, and 95% of queries are delivered within 10 hops.



Attacker (%)	0	20	30	40	50
size of nCert = 7	0 (0)	0.011 (0.013)	0.128 (0.218)	0.781 (1.626)	
size of nCert = 11			0.001 (0.002)	0.025 (0.042)	0.22 (0.49)

Figure 7: Dropping Test

6.4 Optimization

In this section, we discuss how to reduce the latency of Myrmic queries by using recursive routing. Recursive routing has lower delay than iterative routing because fewer message are sent; in addition, recursive routing can utilize *proximity neighbor selection* (PNS), which allows a node to select neighbors (fingers, in Chord's terms) with low *RTT*, resulting in low stretch, the ratio of query delay to the network delay between the querier and the root. This technique can significantly reduce the query delay, for example Dabek *et al.* [22] show that DHTs using PNS can deliver queries with constant stretch, independent of the network size.

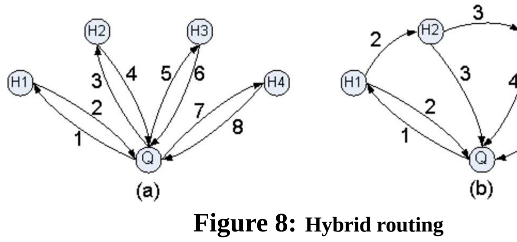


Figure 8: Hybrid routing

We can adapt Myrmic's iterative routing protocol to take advantage of recursive routing. Figure 8 (b) illustrates the basic idea. When an intermediate hop H_i receives a query message from the querier Q, it 1) delivers the message to the next hop H_{i+1} according to recursive routing, and 2) sends $nCert_{H_{i+1}}$ back to Q. Q monitors the query, verifying that each hop makes progress as described in Section 4.3 and verifying the root at the conclusion of the query. If a query ever fails to make progress, Q can pick another node in the nCert of the current next hop, and restart the process. Note that the security analysis of the iterative routing protocol also applies to this optimized protocol because of the progress verification and root verification. This optimization can also utilize PNS, where a node's candidate fingers are limited to the nodes that are close (in the ID space) to the Chord next hop. In case the size of a nCert is large, e.g. $2l + 1 = 15$, a nCert provides enough candidates [22]. With a smaller nCert, intermediate hops may need to return adjacent nCerts (one of which lists the chord next hop) for progress verification. More work is needed to understand the interaction of nCert size, PNS and performance of this optimization.

7. RELATED WORK

Sit and Morris [34] present a taxonomy of possible attacks on DHTs and applications built on them. They further provide several design principles to prevent them. One of the identified denial-of-service attacks, the so called *Rapid Joins and Leaves* attack, which is also referred to as *Churn*, was studied by several groups [35, 33, 36]. Lynch *et. al.* [37] propose to use a Byzantine Fault Tolerance replication algorithm to maintain state information for correct routing – even though this solution is quite elegant, it is too expensive to be used in practice since it requires an agreement between the replicas at each routing step. The Sybil attack has been studied by several groups [38, 25]. Two Sybil-resistant schemes based on social links were recently proposed in [39, 40]. None of these works consider the problem of root verification, leaving them vulnerable to root-spoofing attacks.

The seminal work on DHT routing security is by Castro *et al.* [23]; they propose but do not implement a DHT where each node maintains an optimized finger table for fast routing and a constrained table for "secure routing." When performing a lookup on k, a node first makes an "optimized" query, and performs a test of the result (that involves communicating with all neighbors of $root(k)$). If the test fails, the querier launches many parallel recursive queries using the constrained finger table; if any of these queries reaches any honest replica root, it is broadcast to all other replica roots. Assuming disjoint paths are taken by all queries, the number of queries sent should be $n^{O(\log(\frac{1}{1-f}))}$, that is, polynomial in the number of nodes. Thus *asymptotically*, Myrmic is exponentially more efficient than this scheme while including a proof of security. Concretely, the authors report on simulations showing that when adversaries do not perform certain known attacks, the scheme can deliver queries to 99.9% of keys in a node with 100,000 nodes and 30% compromised nodes using 32 parallel lookups. Using 32 parallel lookups and assuming $f = 0.25$ fraction of adversaries, [23] report that the expected number of messages sent per query is 451, compared to 11 for Myrmic; the reported bit complexity of a query in [23] is 5.6KB + 22KB + 12KB or about 39KB, plus 32 copies of the value stored under k; when optimized for bandwidth (by only sending the nCert of the next hop rather than the entire finger table), Myrmic sends 11 + 7 nCerts, each of which has a 128B certificate, 7 24B (nodeID, IP) pairs, and a 128B signature or 7.6KB; the correct root sends only 1 copy of the value stored for k.

Fiat and Saia [41, 42] give a protocol for a "content-addressable" network that is robust to node removal. Kubiatowicz [43] make Pastry and Tapestry robust using *wide paths*, where they add redundancy to the routing tables and use multiple nodes for each hop. Fiat *et. al.* [20] define a *Byzantine join* attack model where an adversary can join Byzantine nodes to a DHT and put them at chosen places. All of these results require a DHT node to maintain $O(log^2(n))$ links to other nodes, have $O(log(n))$ latency and $\Omega(log^2(n))$ message complexity per query. [20] makes use of a protocol of Scheideler [28] to rotate nodes when they join the network, providing strong guarantees about the density of adversarial nodes without need of a certified identity; this protocol does not, however, defend against Sybil attacks.

In the Eclipse attack [23, 34], or routing table poisoning attack, malicious nodes conspire to fool honest nodes to include the malicious nodes into their routing tables. Singh *et. al.* [44] observe that a malicious node launching an eclipse attack has a higher in-degree than honest nodes. They propose a method of preventing this attack by enforcing in-degree bounds through periodic anonymous distributed auditing. Nodes that fail the test are dropped from the testing node's routing table. Condie *et. al.* [29] mitigate eclipse attacks using induced churn. The main idea includes three components: periodically reset routing tables to constrained ones [23], limit routing table update rate, and periodically change nodes' nodeIDs. We note that the Eclipse attack is not possible against Myrmic because we employ a Chord-style constrained routing table.

8. CONCLUSION

Recently, a significant amount of effort has been devoted to making DHTs more robust against environmental interference, but there has been considerably less work on implementing DHTs that are secure against adversarial behavior. With increasing use of these protocols in economically attractive applications, ensuring correctness and availability of DHT routing is a fundamental requirement. To address problem, we introduced *Myrmic*, a novel DHT routing protocol in this paper. To the best of our knowledge, Myrmic provides the first implementation of a DHT routing protocol that allows root verification (by internal as well as external entities) as well as efficient (comparable to Chord) message delivery even in the presence of a significant fraction of faulty nodes. In many applications, it can be used as a drop-in, secure replacement for other existing DHT routing protocols.

9. REFERENCES

[1] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," in *SIGCOMM*, 2001.

[2] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A peer-to-peer lookup service for internet applications," in *SIGCOMM*, 2001.

[3] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Middleware*, 2001.

[4] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "Opendht: A public dht service and its uses," in *SIGCOMM*, 2005.

[5] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A Resilient Global-scale Overlay for Service Deployment," *JSAC*, vol. 22, no. 1, 2004.

[6] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," in *OSDI*, 2002.

[7] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage," in *ASPLOS*, 2000.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *SOSP*, 2001.

[9] A. Rowstron and P. Druschel, "Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility," in *SOSP*. ACM, 2001.

[10] B. Cohen, "Incentives Build Robustness in BitTorrent," in *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, 2003.

[11] S. Iyer, A. Rowstron, and P. Druschel, "SQUIRREL: A decentralized, peer-to-peer web cache," in *ACM PODC*, 2002.

[12] M. J. Freedman, E. Freudenthal, and D. Mazires, "Democratizing Content Publication with Coral," in *USENIX/ACM NSDI*, 2004.

[13] M. Castro, M. B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman, "An Evaluation of Scalable Application-level Multicast Built Using Peer-to-peer overlays," in *Infocom*, 2003.

[14] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "Splitstream: high-bandwidth multicast in cooperative environments," in *SOSP*, 2003.

[15] K. Park, V. S. Pai, L. L. Peterson, and Z. Wang, "Codns: Improving dns performance and reliability via cooperative lookups," in *OSDI*, 2004.

[16] V. Ramasubramanian and E. Sirer, "The design and implementation of a next generation name service for the internet," in *SIGCOMM*, 2004.

[17] A. Stubblefield, J. Ioannidis, and A. D. Rubin, "A key recovery attack on the 802.11b wired equivalent privacy protocol (wep)," *ACM Trans. Inf. Syst. Secur.*, vol. 7, no. 2, pp. 319–332, 2004.

[18] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs," in *Crypto*, 1998.

[19] R. J. Anderson and R. M. Needham, "Programming satan's computer." in *Computer Science Today*, 1995.

[20] A. Fiat, J. Saia, and M. Young, "Making chord robust to byzantine attacks," in *ESA*, 2005.

[21] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *Sigcomm Comput. Commun. Rev.*, 2003.

[22] F. Dabek, J. Li, E. Sit, J. Robertson, M. Kaashoek, and R. Morris, "Designing a dht for low latency and high throughput," in *NSDI*, 2004.

[23] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach, "Secure routing for structured peer-to-peer overlay networks," in *OSDI*, 2002.

[24] A. Singh, M. Castro, P. Druschel, and A. Rowstron, "Defending against eclipse attacks on overlay networks," in *EW11*, 2004.

[25] J. R. Douceur, "The sybil attack," in *Proc. of the IPTPS02*, 2002.

[26] P. Maymounkov and D. Mazíeres, "Kademlia: A peer-to-peer information system based on the xor metric," in *IPTPS*, 2001.

[27] S. Crosby and D. Wallach, "Denial of service via algorithmic complexity attacks," in *USENIX Security*, 2003.

[28] C. Scheideler, "How to spread adversarial nodes? Rotate!" in *STOC*, 2005.

[29] T. Condie, V. Kacholia, S. Sankararaman, J. Hellerstein, and P. Maniatis, "Induced churn as shelter from routing table poisoning," in *NDSS*, 2006.

[30] I. Osipkov, P. Wang, N. Hopper, and Y. Kim, "Robust Accounting in Decentralized P2P Storage Systems," in *ICDCS*, 2006.

[31] OpenSSL Project Team, "Openssl," http://www.openssl.org/, 2006.

[32] "Berkeley chord library," http://i3.cs.berkeley.edu/.

[33] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a dht," in *USENIX Annual Technical Conference*, 2004.

[34] E. Sit and R. Morris, "Security Considerations for Peer-to-Peer Distributed Hash Tables," in *IPTPS*, 2002.

[35] M. Castro, M. Costa, and A. Rowstron, "Performance and dependability of structured peer-to-peer overlays," Microsoft Research, Tech. Rep. MSR-TR2003 -94.

[36] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek, "Comparing the performance of distributed

hash tables under churn." in *IPTPS*, 2004.

[37] N. Lynch, D. Malkhi, and D. Ratajczak, "Atomic data access in content addressable networks," in *IPTPS*, 2002.

[38] E. Friedman and P. Resnick, "The Social Cost of Cheap Pseudonyms," *J. of Economics and Management Strategy*, 2001.

[39] S. Marti, P. Ganesan, and H. Garcia-Molina, "DHT Routing Using Social Links," in *P2PDB*, 2004.

[40] G. Danezis, C. Lesniewski-Laas, M. F. Kaashoek, and R. Anderson, "Sybil resistant DHT routing," in *ESORICS*, 2005.

[41] A. Fiat and J. Saia, "Censorship resistant peer-to-peer content addressable networks," in *SODA*, 2002.

[42] J. Saia, A. Fiat, S. Gribble, A. Karlin, and S. Saroiu, "Dynamically fault-tolerant content addressable networks," in *IPTPS*, 2002.

[43] K. Hildrum and J. Kubiatowicz, "Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks," in *DISC*, 2003.

[44] A. Singh, T.-W. J. Ngan, , P. Druschel, and D. S. Wallach, "Eclipse attacks on overlay networks: Threats and defenses," in *Infocom*, 2006.

APPENDIX

A. TIMEOUT VALUES

During a query, the current hop may be malicious, dead, or having long RTT. All the three cases may result in a timeout. When considering all of them, the expected number of steps a query take is $\frac{1}{2(1-f-x-y)} \log n$ where x is the probability that the current hop is dead and y is the probability that the current hop having long RTT than the timeout value. The value of x is usually small since it is the probability that a node dies between two *fix routing table* messages send to it from another node. Hence we only consider f and y. Denoting the timeout value as t_o, the query delay is bounded by $\frac{1}{2(1-f-y)} \log n \times t_o$. On the other hand, a small timeout value may need more bandwidth. Converting malicious nodes to blackholes, the extra bandwidth overhead due to a specific timeout value is $\frac{1}{(1-y)} - 1$.

B. LIFETIME OF NCERTS

The life time of a nCert, denoted as T_{ncl}, represents a tradeoff between security and efficiency. Intuitively, a longer T_{ncl} gives improved efficiency since less nCerts need to be renewed. On the other hand, a shorter T_{ncl} is more secure since less (honest) nodes leave during this period. Suppose a node listed in a nCert left, then the nCert is revoked. In this case, from both security and efficiency points of view, we prefer that the nCert expires soon after its revocation since it needs not be renewed. The same argument applies for joins too. We compute the probability of the event e that some nodes in a nCert leave or some nodes join in the range covered by the nCert during its life time: (Note that this is also the percentage of nCerts revoked before they expire.)

$$P(e) = 1 - (1-\alpha)^{2l+1} \times (\frac{n-(2l+1)}{n})^{n \times \alpha} \quad (4)$$

After selecting system parameters (i.e., α and $2l+1$) as shown in section 6.1, one can compute the percentage of nCert update operations (renewing nCerts) due to these parameters using equation 4. For example, suppose $n = 5000$, $\frac{T_{ncl}}{T_{ms}} = 0.1$, and $2l+1 = 7$, the percentage of nCerts revoked before they expire is 63%. I.e., 63% of nCerts are renewed due to network churning and 37% of nCerts are renewed due to the parameter we selected.

C. NA REPLICATION

To increase availability and remove the single point of failure, one may replicate the NA. This task is easy in most cases, since the replicas of the NA only need to share a private key to sign nCerts. In case when two nodes, X and Y, join the same neighborhood at the same time and their join requests are sent to two different NA replicas NA_1 and NA_2, there exists a subtle synchronization issue. If NA_2 is not aware of X and NA_1 is not aware of Y, then nCerts issued by the two NAs may have conflicting ranges.

The NA replication works as follows. The ID ring is divided into pieces, called zones, and each NA replicas is responsible to one zone. All the NA replicas shares a MAC key to authenticate messages among them. A node's nCert update request can be sent to any of the NA replicas. Receiving a request, the NA replica, say NA_1, forwards the request to the one responsible to the nodeID, say NA_2. NA_2 processes the request following the nCert update protocol shown in section 4.2. If the neighbor list NA_2 built includes nodes in an other NA replica's zone, say NA_3's, NA_2 notifies NA_3 to lock the border between them. NA_2 releases the lock after processes the request. Before NA_2 releases the lock, NA_3 processes requests to which it is responsible as usual, except that it holds the process of a request if the neighbor list it built for this request includes a node in NA_2's zone. The messages sent among NA replicas are acknowledged. If a acknowledge, e.g. from NA_1 to NA_2, is missing, NA_2 pings NA_1 to find out if NA_2 is offline. If so, NA_2 notifies others NA replicas and redivides the ID space. Similar procedure is followed if a NA replica holds a lock for too long. Note that since ID space is a ring, the zone of a NA replica has two borders. As long as a neighbor list built for a nCert update request does not cross both of the borders, the locking scheme does not cause a deadlock. I.e., A deadlock does not happen if every zone has at least $4l+1$ honest nodes. Hence deadlock is not a problem in practice, since $4l+1$ is much smaller than the number of nodes a NA server can support even with a extreme high churn rate.