

# Myrmic: Secure and Robust DHT Routing

## 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 need 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 can be used to construct a wide range of distributed applications, for example P2P file system [6], P2P archival systems [7, 8], P2P DNS [9, 10], and Resilient overlay networks [11].

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 are not designed to deal with *adversarial* faults that maliciously prevent nodes from discovering the correct mapping between identifiers and peers. Violating the correctness of this mapping can in turn invalidate the correctness or security of protocols running on top of the DHT, since they assume the mapping to be 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 [12, 13, 14]. Unfortunately, while many of these provably secure schemes scale well asymptotically (for example the scheme in [15] 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 [16] 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 30% of nodes behave maliciously by dropping all routing requests. Thus, it can be used as a drop-in, secure replacement for Chord.

Four 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  $\delta$  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. 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. While this property is not crucial to the design of our system it simplifies some aspects of application design, for example, allowing new nodes to join the DHT without the additional trust assumptions of a trusted gateway or the additional communication cost of using several redundant gateways. Finally, Myrmic makes use of adjustable *soft timeouts* to provide a trade-off between query response latency and bandwidth while guaranteeing reliable delivery of messages in the face of transient or malicious faults.

The remainder of this paper is organized as follows. Sec-

tion 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  $(k, v)$ , the node  $root(k)$  stores the pair, where the value  $v$  is application-specific information. When a node (a querier) 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  $p^1(x), p^2(x), \dots$ ) and a list of its nearest successors in the *Id space*, denoted by  $s^1(x), s^2(x), \dots, s^i(x)$ . We follow Chord in using this *constrained* routing table; for a discussion of the performance implications of this decision, and possible optimizations, see Section 6.5.

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 DHT

Like other networks, DHTs are vulnerable to attacks. Below, we briefly overview the attacks (specific to DHTs) proposed in previous work [17, 18] and the known approaches to dealing with them.

**Sybil attack** [19]: 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 [17], 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** [17]: 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, 20].

**Routing Table Poisoning (Eclipse Attack)** [17]: 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 having flexibility in selecting routing table entries.

**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 An Attack Scenario

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. Following the same attack logic, malicious nodes can attack insertion messages as well as DHT control protocols, such as routing table maintenance or joins. The reason is that all of the DHT protocols rely on the ability to find the correct root of a key. 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 disrupt most of the functionalities of the DHT.

### 3.2 Problem Description

Security of DHT routing is thus a weak link in building secure DHT-based applications. Conversely, combining a secure DHT routing protocol with existing security techniques will allow us to construct secure DHTs and DHT-based applications. Castro *et. al.* [17] 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 attacks [21]. 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** ( $\delta$ -SECURE DHT ROUTING). A routing protocol is  $\delta$ -secure if it ensures that with probability at least  $1 - \delta$ , when a non-faulty node  $A$  initiates a lookup for a uni-

formly 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 malicious nodes.

Notice that the above definition explicitly states that we will *allow* a certain fraction,  $\delta$ , of queries to fail. Thus an important question is what values of  $\delta$  a protocol can support. Later in this paper, we show that Myrmic can deliver 99.99% of packets within 1.5 times the latency of Chord even with 25% of attackers dropping packets. (i.e. if Chord needs 5 hops on average without an attacker, Myrmic needs 7.5 hops even with 25% of nodes behaving maliciously) 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.

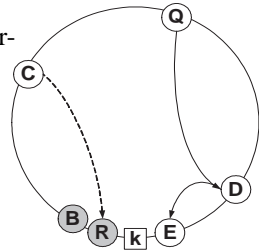
### 3.3 Assumptions

Before introducing Myrmic, we discuss a few assumptions. To avoid attacks related with nodeId assignment such as the Sybil attack [19], we assume the existence of an off-line certification authority as in [17]. 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, adversaries have complete control of IP- and DHT-layer traffic passing through faulty nodes. We assume that all nodes are loosely time synchronized, e.g. honest nodes’ clocks agree to within a few seconds.<sup>1</sup> 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)*.

<sup>1</sup>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



**Figure 1:** An attack scenario.

The *NA* only participates in DHT network management by issuing *Neighborhood Certificates (nCerts)* to a small set of nodes after DHT membership events such as join and leave. The *NA* is *not* involved in any other functionality of DHT routing, in particular queries can proceed without contacting the *NA*. The *NA* has a public/private key pair for signing certificates and we assume that its certificate is publically available. The *NA* is stateless and can be replicated to handle high churn rates or transient node failures. Similar to the CA, the *NA* is a central point of trust rather than a central point of failure. Thus 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. Notice that 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 individual nodes' current ranges using nCerts issued by the *NA*. This prevents a malicious node from impersonating the root of a key outside its range or routing a query to an incorrect node.

A naive 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 one can 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 range. 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 usually the replica roots, (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.,  $range(R) = (p(R), R]$ . Hence by including both  $R$  and  $p(R)$  into  $R$ 's nCert,  $R$  can prove its range in a DHT with static membership. With membership changes, 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 the freshness of a nCert. If any witness can prove that a revoked nCert is indeed revoked, then a malicious node can use a revoked nCert only if all (live) witnesses are its colluders. 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 the nearest neighbors of the nCert's owner node to be the witnesses as they must maintain this information anyways and may also be replica roots, depending on the application. The format is thus:

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

A nCert is signed by the *NA* using secure digital signature  $Sign_{sk_{na}}(\cdot)$  with private key of the *NA*. The nCert also includes its issue time and its expiration time. The  $nList_R$  in  $nCert_R$  includes tuples  $I = (nodeId, IP)$ , which allow direct IP connections to  $R$ 's neighbors. The size of the  $nList$  ( $= 2l + 1$ ) is a system parameter defined by the *NA*. We explain how this parameter effects security in section 5.

Figure 2 summarizes the root verification procedure, which

```

// verify if R is the root of key k
Q.verify_root(nCertR, k)
if(nCertR.expireTime < currentTime or
  is_root(nCertR, k) is false)
  return false;
else
  for(X ∈ nCertR.nList)
    nCert'R = X.find_nCert(R);
    if(nCert'R.issueTime > nCertR.issueTime
      and is_root(nCert'R, k) is false)
      return false;
  return true;

// verify if R is the root of k according to nCertR
Q.is_root(nCertR, k)
if(k ∈ (nodeIdp(R), nodeIdR])
  return true;
return false;

```

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

assumes that all nodes have current nCerts. Here a querier  $Q$  uses  $nCert_R$  to verify whether  $R = \text{root}(k)$  by applying two tests. First we check that  $k \in \text{range}(R) = (p(R), R]$  where  $p(R)$  and  $R$  are included in  $nCert_R$ ; second, we check that  $nCert_R$  is fresh. This is accomplished by obtaining copies of  $nCert_R$  directly from the witnesses. If a witness  $X$  gives a valid  $nCert'_R$  which has issueTime later than the issueTime in  $nCert_R$  and the root of  $k$  according to  $nCert'_R$  is different from  $R$ , then  $R$  fails the test. The querier  $Q$ 's communication overhead is to contact  $2l$  witnesses and the computation overhead is to verify the  $2l$  signatures of the replied nCerts.

## 4.2 Neighborhood Certificate Update

At 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  $NA$  updates the certificates of every node whose neighborhood changes because of this node addition. More specifically, the  $NA$  updates the nCerts of all the nodes listed in  $nList_J$  to include  $J$  in their  $nLists$ . Similar updates are also carried out when the  $NA$  is notified that  $J$  has left the network.

When a new node joins a Chord network, it first learns its neighborhood information from a bootstrap node, and then gradually fills in its finger table using queries to the appropriate Ids. Our join protocol only modifies this first portion of the joining protocol but not the other parts, 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 bootstrap node  $B$ , the joining node  $J$  locates the node  $R = \text{root}(\text{nodeId}_J)$  using the secure iterative routing protocol (to be presented in section 4.3). Next  $J$  contacts  $NA$  to get  $nCert_J$ .

To generate  $nCert_J$ ,  $NA$  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. For this purpose,  $NA$  obtains and verifies  $R$ 's nCert using the root verification protocol. Once  $NA$  has  $R$ 's nCert, it

```

// node J joins the network. node B is used for bootstrap
J.join(B)
R = B.find_root(J);
NA.update_nCerts(R, J);
init_finger_table(B);
update_others();

// issue a nCert to the joining node and update its neighbors' nCerts
NA.update_nCerts(R, J)
if(accept(J) is true and verify_root(nCertR, J) is true)
  list = nil;
  list = construct_neighbor_list(R, J);
  generate_distribute_nCerts(list);

// NA constructs a list of the nearest neighbors of the joining node
NA.construct_neighbor_list(R, J)
list = nil;
for(X ∈ R.nCert)
  for(Y ∈ X.nCert)
    for(Z ∈ Y.nCert)
      if(in_list(list, Z) is false) and ping(Z) is live)
        insert_into_list(list, Z);
insert_into_list(list, J);
while(count_live_successors(list, J) < 2l)
  get_more_successors(list, J);
while(count_live_predecessors(list, J) < 2l)
  get_more_predecessors(list, J);
return list;

NA.generate_distribute_nCerts(list, J)
nListJ = gen_nList(list, J);
for(X ∈ nListJ);
  nListX = gen_nList(list, X);
  nCertX = Signskna{nListX, issueTime, expireTime};
  send(nCertX);

// initialize finger table of the local node
// this procedure is not modified from chord
J.init_finger_table(B)

// update other nodes' finger table
// this procedure is not modified from chord
J.update_others()

```

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

can directly contact the neighbors of  $R$ . Next  $NA$  calls  $\text{construct\_neighbor\_list}()$  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:  $\text{list} = \{\dots, I_{p^{2l}(J)}, \dots, I_{p(J)}, I_J, I_s(J), \dots, I_{s^{2l}(J)}, \dots\}$ . This list includes all the information  $NA$  needs to generate new nCerts. Once  $NA$  has the list, it calls  $\text{generate\_distribute\_nCerts}()$  to generate  $2l + 1$  nCerts of  $J$ , its nearest  $l$  predecessors, and its nearest  $l$  successors. Each  $nCert_X$ , is sent all of the nodes listed in it (including  $X$ ).

```

// node X maintains updated neighborhood
X.maintain()
for(Y ∈ nListX)
  if(ping(Y) is dead)
    Z = Y;
    do
      Z = find_root(X, (Z.id + 1));
      while(ping(Z) is dead)
        NA.update_nCerts(Z, Z);

```

**Figure 4: The pseudocode for a node X maintains updated neighborhood**

When a node *leaves* the DHT, other nodes (gradually) update their state tables. The range of the left node must be

allocated to its neighbor(s). Hence once a leave is detected, the *NA* should be notified to update nCerts of neighbors of the left node. In the proposed scheme, a node *X* periodically calls *maintain()* to ping nodes listed in its nCert. If it believes 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<sub>Z</sub>* (including *X*) obtain updated nCerts.

We now briefly consider the additional overhead of *update\_nCerts*. To simplify the discussion, we consider the number of messages an entity sends as its communication complexity and digital signature operations as its computation complexity. Note that nodes in neighbors' nCerts are overlapping and *nCert<sub>X</sub>* is stored by all nodes listed in it. Hence, in the three *for* loops of *update\_nCerts()*, *NA* can pull the  $4l + 1$  nCerts from the  $2l + 1$  nodes listed in *nCert<sub>R</sub>*. *NA* pings another  $2l$  nodes in *count\_live\_predecessors(list, J)* and *count\_live\_successors(list, J)*. *NA* also signs the new generated  $2l + 1$  nCerts. Since a nCert is sent to all nodes listed in it, *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

Given the root verification procedure and securely maintained finger tables, we can construct an iterative routing procedure that correctly and quickly routes DHT lookups by verifying that intermediaries give correct next hops and that the alleged result of a query for *k* is the correct *root(k)*. Figure 5 shows the procedure a Myrmic client *Q* uses to route a query for key *k* through a possibly untrusted gateway *G* (which may be *Q*).

The main idea is as follows: *Q* maintains a circular list *ring* that is *Q*'s current view of the DHT. The *ring* is populated with nodes and their nCerts polled from nodes that *Q* has contacted during the query. *Q* repeatedly finds the next node to contact from the ring until it finds the root.

At any point during the query, *Q* considers some node *current* to be the next node on the route. *Q* contacts *current* and asks for the nCerts of each of *current*'s neighbors and each of its fingers. If *current* does not provide nCerts consistent with a complete finger table, or it does not respond to this request, it can be ignored as a faulty node. Otherwise these nodes' nCerts are checked<sup>2</sup> and the nodes listed in them are inserted into *ring*. If *current* is *root(k)* or any nCert received from *current* contains *root(k)*, the lookup is completed. Notice that we therefore need only to find a live, honest neighbor of *root(k)* in order to correctly complete a lookup.

If *current* is not *root(k)*, *Q* needs to find the next node to

<sup>2</sup>Note that *Q* does not need to contact any node to verify these intermediate nCerts- only signature verification is required, which shows that the nCert was recently issued by the *NA*.

```
// find the root of k using gateway node G
Q.find_root(G, k)
nhops = 0;
ring = nil;
current = G;
do
  for(nCertX ∈ {nCert stored by current})
    if(is_root(nCertX, k) is true)
      if(verify_root(root, k) is true)
        return root;
    else
      insert(ring, nCertX);
      mark_as_contacted(current);
      P = find_predecessor_on_ring(ring, k);
      S = find_successor_on_ring(ring, k);
      if(P is nil and s is nil)
        return nil;
      if(|nodeIdS - k| < |nodeIdP - k|
        and |nodeIdS - k| < θ)
        current = random_select_from(nCertS);
      else
        current = random_select_from(nCertP);
      nhops ++;
  while(nhops < NHOPS_MAX);
return nil;
```

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

contact. Thus *Q* finds nodes *P* and *S* whose nodeIds immediately precede and succeed *k* on the *ring*. Unlike Chord, which uses *P* as the next hop, our next hop decision depends on *S* and *P*'s neighbors. First, if the distance between *S* and *k* is smaller than a threshold  $\theta$ , then *Q* sets a random neighbor of *S* as the next node to contact, where the threshold  $\theta$  is *Q*'s estimate of the distance between *root(k)* and its furthest succeeding witness. Hence, succeeding witnesses can be used not only to verify a nCert but also to find the root. In case *S* is not used, *Q* finds a nCert that contains *P* and randomly picks one of the nodes listed in that nCert as the next node to contact. This random selection process strengthens our protocol against an attacker attempting to provide nCerts that list colluders as the deterministic next hop.

The protocol fails (returning *nil*) if all nodes on *ring* have been marked as contacted or *nhops* is larger than a threshold *NHOPS\_MAX*, where *nhops* is the number of nodes that *Q* has tried to contact during the query. *NHOPS\_MAX* is a system parameter computed based on the anticipated node join-leave rate and percentage of malicious nodes. Assuming *NHOPS\_MAX* is sufficiently large, the protocol may fail only if *Q* has contacted all nodes, to which it has a path and none of them know *root(k)*. This failure may be caused by two reasons: either *Q* contacts all nodes in the DHT and cannot find a valid nCert; or the DHT is partitioned by malicious nodes and dead nodes, where *Q* and *root(k)* are in different components. This can happen only when the network has a high percentage of malicious nodes and/or the churn rate is extremely high.

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 *current*, and simply contacts another neighbor of either *S* or *P* according to the next hop selection process de-

scribed above. If  $Q$  receives a message from *current* after a soft timeout, it will still process the result and update *ring*. When all nodes on *ring* have been marked as contacted and the *hard timeout* for each node is reached,  $Q$  determines that the lookup has failed and drops the message. Using a short soft timeout may increase the number of messages sent, while reducing the delay caused by a malicious node who does not respond, or a slow link. We note that as the soft timeout approaches 0, this lookup process becomes similar to the parallel lookup process employed by Kademlia [20] (since we contact several next hops in close succession) and can decrease the lookup time. On the other hand, by using a longer soft timeout, we can reduce the total number of lookup messages sent; this gives the node  $Q$  the opportunity to trade off lookup latency for bandwidth consumption.

## 5. SECURITY ANALYSIS

In this section, we argue briefly for 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 very 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. 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,<sup>3</sup> is at most  $f^{2l}$ . Thus with high probability the *NA* discovers all neighbors of  $n$  and issues a

<sup>3</sup>for purposes of correctness, we treat faulty nodes and adversarial

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  $\delta$  be a "security parameter" for secure DHT routing, e.g. the probability of routing failure we are willing to tolerate.<sup>4</sup> 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 *current* in the iterative routing step,  $Q$  requests the finger table for *current* and the nCert of each finger. Thus  $Q$  can check that the relevant fingers returned by *current* are in fact the nodes responsible for the keys  $nodeId(current) + 2^i$  for  $i \in \{1, 2, \dots\}$ . If they are not, the node *current* 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.

Now, we define the *chord next hop* from node  $n$  to key  $k$  to be the next node after  $n$  that (iterative) Chord would query in a fault-free ring. 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 nodes identically, i.e., a node that goes offline during the duration of an nCert with probability  $f$  is considered to be faulty in this analysis

<sup>4</sup>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.

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  $f^{2l+1}$ , we see by the union bound that the probability of such failure is at most  $f^{2l+1} \log n$ . Thus setting the neighborhood size  $2l + 1 = \left\lceil \frac{1}{\log(1/f)} \left( \log \log n + \log \frac{1}{\delta} \right) \right\rceil$  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  $\sum_{i=1}^{2l+1} i \cdot (1-f)^{i-1} \leq (1-f) \sum_{i=1}^{\infty} i \cdot f^{i-1} = 1/(1-f)$ . 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. This is supported by our experimental results.

## 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” [22, 23, 24] 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 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 independent components: the DHT node and the *NA*. Both components are implemented in C and use Openssl 0.9.8 [25] for RSA digital signature with SHA-1 cryptographic hash function. The Myrmic-DHT node component is built upon i3-Chord implementation [26]. *NA* is multithreaded to allow updates to continue without blocking for other update’s network responses. The whole implementation consists of approximately 8,500 lines of code including 3,300 lines of i3-Chord code, 4,000 lines of Myrmic DHT node code, and 1,200 lines of *NA* code. For evaluation, a simple DHT query application is built on top of the DHT overlay. The DHT layer routes each query to the root and performs other maintenance functions such as routing table updates. We evaluate the performance of our prototype implementation 1) on PlanetLab [16] to evaluate performance on wide-area network and 2) on a local testbed to assess the robustness of Myrmic against message dropping attack.

### 6.1 Parameter Selection

Below we discuss how to select three parameters: nCert life time, nCert size, and timeout values.

### 6.1.1 Life Time 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. To find an appropriate  $T_{ncl}$  value that provides high security and has low overhead, we show the interaction between a  $T_{ncl}$  value and median node session times as well as nCert size. The median node session times can be found in published studies. For example, Rhea *et. al.* [27] surveyed published studies of observed median session times of deployed file-sharing P2P networks.

We model churn as previous works such as [27]. The times of nodes' leaves follow a Poisson process, with an event rate  $\lambda$ . Nodes joins follow the same process resulting in a stable network size. The event rate can be computed based on median node session time

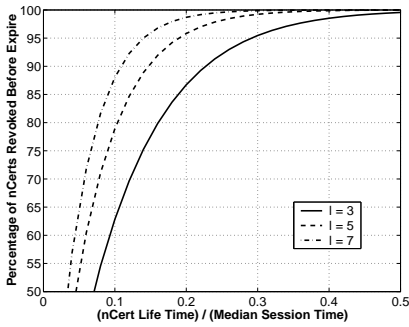
$$\lambda = \frac{n \times \ln 2}{T_{ms}} \quad (1)$$

where  $T_{ms}$  denotes a median session time. Hence the fraction of nodes join/leave the DHT during a  $T_{ncl}$  is

$$\alpha = \frac{n \times \ln 2}{T_{ms}} \times \frac{T_{ncl}}{n} = \frac{\ln 2 \times T_{ncl}}{T_{ms}}. \quad (2)$$

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.)

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



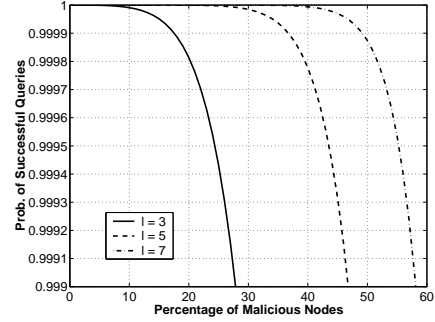
**Figure 6:** Percentage of nCert revoked before expired ( $n = 1000$ )

Figure 6 shows the interaction between nCert size, the ratio between nCert life time and median session time, and the percentage of nCerts revoked before they expire. For example, when  $l = 3$  and the ratio between nCert life time and median session time is 0.16, the percentage of nCerts revoked before they expire is 80%. In other words, only 20% of nCerts need to be renewed. Therefore we increase the load of *NA* by 20% of natural churning. Note renewing nCerts does

not cause the problems caused by churning such as dead finger table entries or missing replicas of data items. In the rest of this section, when computing other parameters, we set the percentage of nCerts revoked before they expire as 80%.

### 6.1.2 Size of nCerts

As the case of the life time of a nCert, the size of a nCert ( $2l + 1$ ) is also a tradeoff between security and efficiency. We compute  $l$  based on the required security level, churn rate, and nCert life time. The security level is measured as the percentage of queries that successfully identify the roots. The probability that a queries is sent to a malicious node who happens to be able to use a revoked nCert is  $f \times (f + \alpha)^{2l}$ , where  $\alpha$  is computed with equation 2.



**Figure 7:** Prob. of Successful Queries

Figure 7 shows this probability while varies the percentage of malicious nodes. Suppose the required security level is 99.99%, with  $l = 3$ , Myrmic can handle  $f = 17\%$ . With  $l = 7$ , Myrmic can handle  $f = 48\%$ .

### 6.1.3 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  $\frac{1}{2(1-f-y)} \log n \times t_o$ .

We measured the pairwise ping time of PlanetLab nodes. Each node sends 10 pings to every node in our experimental setup. The average, median, 75% and 99% were 54, 50, 78, and 177 *ms*. In our implementation, we use 78 *ms* as soft timeout and use 200 as timeout value of nCert verification.

## 6.2 Performance Analysis

While we have analyzed the cost of each protocol in Section 4, we briefly summarize the expected overall costs of Myrmic in terms of computation, bandwidth, and delay.

**Delay** As discussed in Section 5.3, the expected number of “get routing table” (GRT) requests is  $\frac{1}{2(1-f)} \log n$ , when the

timeout is set to be sufficiently large. (for example, the 99th percentile of UDP message delay) If  $f = 0$ , this equals  $\frac{1}{2} \log n$ , which will be used to count the number of signatures and measure the bandwidth overhead below.

**Computation** Digital signature verification is the most computationally expensive operation. During a query,  $Q$  verifies one signature before each GRT request and  $2l + 1$  signatures during root verification. Therefore,  $Q$  is expected to verify  $\frac{1}{2} \log n + 2l + 1$  signatures. However, as we will show later, signature verification time is very small compared with any communication delay, and it does not contribute much to the total delay.

**Bandwidth** To simplify the discussion, we only count the upload bandwidth. The most expensive operation in terms of bandwidth consumption is GRT reply since it includes a routing table. Theoretically, the size of a GRT reply should be  $(\log n + 2l + 1) \times \text{sizeof}(nCert)$ . However, in our protocol, it is less than this value, since the nodes in the routing table are overlapping with witnesses.

### 6.3 Wide-area Evaluation

**Experimental setup.** Each of our wide-area experiments was run on approximately 120 PlanetLab machines<sup>5</sup> as Myrmic nodes, without using the Sirius calendar service [16], and with 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.

Based on this experimental setup, we present wide-area measurements of the *query response time* of Myrmic with different parameters including query rate, size of nCerts, and timeout values (denoted as  $r$ ,  $w$ , and  $t$  respectively, where  $w = 2l + 1$ ). In each experiment, 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 issue queries periodically.

The query response time is 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 application layer and  $t_f$  is the time when the DHT node reports the query result to the application layer after completing the nCert verification. This response time represents the total query time. As discussed before, query and insertion messages can be piggybacked on root verification messages since replica roots are usually the root’s neighbors, included in  $nCert_{root}$ . To simplify the discussion, we assume the set of replica roots and nodes included in  $nCert_{root}$  are the same.

In our iterative routing protocol,  $Q$  approaches the root by polling routing tables from the intermediate hops. We refer to the messages that  $Q$  sent to the hops as *get routing table* (GRT) requests. The number of “steps” a query takes is usually the number of GRT requests plus one (the root

<sup>5</sup>All of these nodes were located only in North America. We expect that they have relatively uniform delay and less non-transitive connectivity [28], which simplifies the analysis of the prototype implementation.

	Signature Generation			Signature Verification		
	Avg	median	99%	Avg	median	99%
PlanetLab	45377	25361	345341	890	158	11691
Local test bed	3473	3282	5712	73	68	100

**Table 1: Digital Signature Time (microsecond)**

verification) step if the soft timeout value is large. If the soft timeout value is small, the reply to a GRT request may come after the timeout, which results in another “trace” of the query in our current implementation. In such cases the number of GRT requests is larger than the number of steps.

**A Few Primitive Operations** Compared with existing implementations, Myrmic requires digital signatures. Since each Myrmic process needs to share a single machine with many other applications on a single PlanetLab machine, we first compare digital signature time on PlanetLab nodes to signing time on our local machines, measuring the wall clock time of each operation. In this experiment, 5 machines are chosen from each testbed. Each of the machines signs and verifies 3000 nCerts. As illustrated in Table 1, the median signature generation time on PlanetLab is 7.7 times longer than on the local machine, while the median verification time is 2 times longer.

**Overall Performance** This section consider the overall performance of Myrmic. In this experiment, we use nCert size  $w = 7$ , and timeout  $t = 78ms$ . Each node sends 500 queries, making the total number of query messages about 60,000. For all evaluations, we set message sending rate  $r = \frac{1}{3}$ , meaning that every node sends 1 message per 3 seconds.

Figure 8 (a) shows the query response time. The 97th and 90th percentiles are  $346ms$  and  $281ms$ . 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 addition, queries can be answered locally, and insertions can be propagated by periodic synchronization between replica roots. Hence, these queries can be handled locally without sending messages. 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 GRT requests one or more times. We categorize the queries based on the number of GRT 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 8 (b), 93% of messages are delivered within 6 GRT requests (not including the root verification message). Figure 8 (b) also shows min, median, and 97th percentile of query response time in each category. The medians are linear to the number of GRT requests approximately. This follows our expectation since Myrmic is not optimized in term of per-hop delay using, e.g., proximity neighbor selection. (See section 6.5 for discussion about such optimizations).



Figure 8: Overall performance with parameters:  $w = 7; t = 78ms$

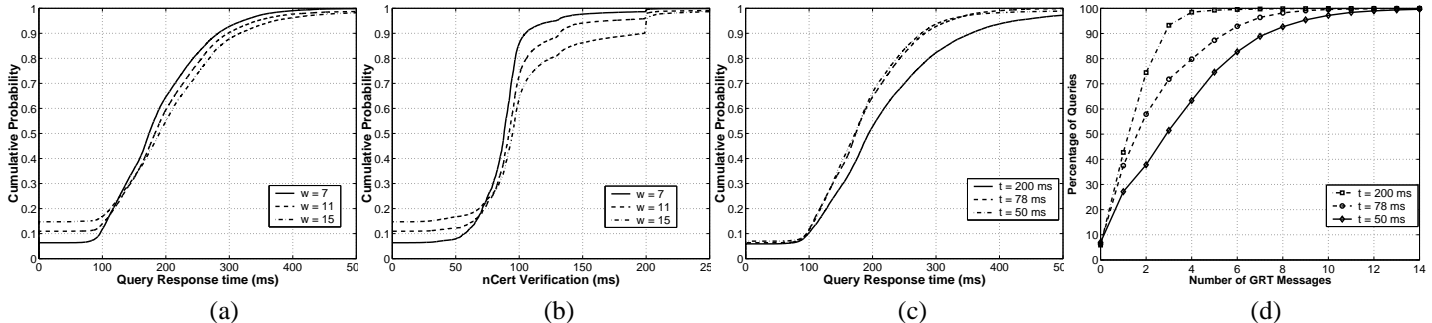


Figure 9: (a) and (b) show effect of nCert sizes when  $w=7, 11, 15; t=78ms$ . (c) and (d) show timeout effect when  $w=7; t=200ms, 78ms, 50ms$ .

**Effect of nCert Sizes** In this experiment, we measure the query response time as well as the *root verification time* with different nCert sizes. The root verification time is defined as  $(t_e - t_s)$  where  $t_s$  is the time when  $Q$  receives  $nCert_{root(k)}$ , and  $t_e$  is the time when  $Q$  completes the root verification. nCert verification time depends on the nCert size as well as the number of neighbors of the root. Figure 9 shows the performance of Myrmic under three nCert sizes:  $w = 7, 11, 15$ , where nCert verification timeout is  $200ms$ , which is larger than the 99th percentile of pairwise ping time,  $177ms$ . Figure 9-(a) shows that the CDF of query response time with different nCert size is close. The main reason causing the difference is nCert verification time as shown in Figure 9-(b). The 90th percentiles with  $w = 7, 11, 15$  are  $105, 131,$  and  $200ms$ . This is because as  $w$  increases, the probability that a neighbor is far away from the querier increases.

**Effect of Timeout Values** One of the most interesting parameters of Myrmic is the soft timeout. As discussed in Section 4.3, a shorter soft timeout will increase the number of GRT messages while increasing a “parallel routing” effect resulting in smaller query delay. This effect is shown in Figure 9 (c) and (d). We set the timeout to be  $50, 78,$  and  $200ms$ , which are median, 75% and 99.9% of the pairwise PlanetLab ping time. Using our analysis given in Section 6.1.3, timeout equal to 50% and 75% pairwise ping time will increase the number of GRT messages by 100% and 33%. Figure 9 (d) supports this analysis. Query response time is decreased when we use shorter timeout values as shown in Figure 9 (c). One interesting result is that query response time with  $50ms$  (50%) and  $78ms$  (75%) timeout looks almost same. This is because of the particular planetlab topol-

ogy, which follows uniform distribution at least until 75% pairwise ping time. However, when we imagine the situation when 75% pairwise ping time takes more than  $100ms$ ,  $50ms$  timeout will be expected to have shorter query response time.

**Evaluation of the NA** An important evaluation of Myrmic is to find the churn rate the NA can handle. We run a set of experiments, each with a different churn rate to find the capacity of the NA. We consider the NA to reach its limit if a higher churn rate causes the join time to increase significantly. In each of this experiment, 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. The highest rate our NA (running Ubuntu Linux with a 3GHz Pentium IV CPU and 1GB RAM) can handle is 30 events (15 joins and 15 leaves) per second (i.e. Churn rate is 15 using the definition of Churn rate in [27]). This number follows our expectation. Note that in the nCert update protocol,  $w = 7$  signatures are generated and 14 signatures are verified (on average). According to Table 1 the computation time for all the signature generation/verification per nCert update protocol is about  $25.3ms$  on average. With churn rate 15, the total signature operation time is  $759 = 2 \times 15 \times 25.3ms$  per second. The median session time reported in [27] ranges from 1 minute to 1 hour. Plugging these session times into Eq. 1, we find that one NA can handle a total number of nodes ranging from 1299 to 77922. 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 multi-

processor systems an optimized NA should be capable of handling a significantly higher workload. Finally, we note in Section 6.5 that it is possible to replicate the NA between nodes to further increase scalability; since the probability of conflicting joins is small, the performance is expected to scale well with processing power.

## 6.4 Local Network Evaluation

One of the most important experiments we wanted to run was when attackers drop GRT requests. Since we want to run on a larger network (1,000 nodes), While PlanetLab might show some interesting behavior, we decided to run this experiment on the local lab to see only the number of GRT requests sent while increasing the percentage of attackers. The lab consists of 36 PCs, each of which runs approximately 30 nodes. Figure 10 shows the result when we have 0, 10, 20, 30 % attackers, who drop all query messages. Even with 30 % attackers. Myrmic dropped only 122 messages out of 1,000,000 messages.

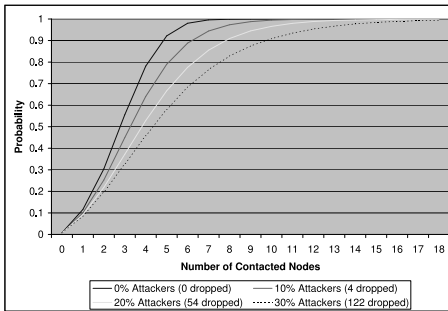


Figure 10: Dropping Test

## 6.5 Discussion

**Optimization** We consider the possibility of using popular techniques to optimize DHT routing in term of query delay. *Proximity neighbor selection* allows a node to select neighbors with low *RTT*, resulting in low stretch. This technique can significantly reduce the query delay. Recent research [29] shows that DHT can deliver queries with constant stretch independent to the network size. However, it is not clear how one can apply this technique while maintaining provable security.

One simple solution to reduce the query delay, similar to the one suggested by Castro *et. al.* [17], is as follows. Each node maintains two routing table. In addition to the constrained routing table used in this paper, the second routing table can be filled using proximity neighbor selection. A query is routed using our protocol for security. In parallel, it is also routed using recursive routing for performance. Note that the root verification protocol works with both iterative routing and recursive routing. Hence if the root returned by recursive routing passes root verification, the query finishes. In this case, routing delay is reduced. On the other hand, suppose the recursive query is dropped by a malicious nodes. Although the secure iterative routing can find the root, the resources used in recursive routing are wasted. One can also combine recursive routing and iterative routing as follows:

When an intermediate hop  $I$  receives a query message from  $P$ , it 1) delivers the message to the next hop  $N$  according to his recursive routing, and 2) sends its nCert back to the querier  $Q$  so that  $Q$  can find the next hop if  $N$  does not respond with its nCert within timeout. In summary, this hybrid routing uses iterative routing to diagnose failed recursive routing.

**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  $NA$ s may have conflicting ranges. A straightforward way of handling this kind of cases is to serialize the join requests sent to the same neighborhood.  $NA_1$  and  $NA_2$  inform each other about the request it currently serves. This approach, however, requires  $m - 1$  messages between the replicas per join where  $m$  is the number of  $NA$  replicas. In addition, there is a chance, although small, that  $NA$  replicas generate conflict nCerts, if any of the messages between  $NA$  replicas is lost. Hence, the problem is how to allow  $NA$  replicas detect and resolve this situation effeciently and reliably.

Our solution is as follows. The  $NA$  replicas must somehow communicate with each other to avoid conflict nCerts. Instead of communicating directly using network messages, the replicas can use the DHT nodes as a communication media. Whenever a  $NA$  replica is handling a join within a neighborhood, it stores a signed token to every DHT nodes in this neighborhood. The token includes information about the joining node and the  $NA$  replica itself. When another  $NA$  replica needs to work on the same neighborhood. It will see the tokens provided by the honest nodes in the neighborhood. Hence as long as one node is honest in the neighborhood, the second  $NA$  replica will notice the potential problem. It can also find the information about the first  $NA$  replica, then handle the problem properly. Once nodes in the neighborhood get new nCerts, they delete the tokens. Note in this solution, (1)  $NA$  replicas communicate directly only if they are handling join at the same neighborhood at the same time; (2) A malicious node cannot use a token to affect the second  $NA$  replica once a new nCert is distributed, since a fresh nCert, which is provided by a honest node in the neighborhood to the second  $NA$  replica and includes the new joining node, can show that the token is stale; (3) No additional message is necessary to store the tokens since they can be piggybacked on other protocol messages; (4) As a communication media, the DHT nodes are more reliable since the tokens are stored at the whole neighborhood. Even if some messages are lost, the  $NA$  replicas can still find the token as long as one honest node in the neighborhood can provide the token to  $NA$  replicas.

**Handling Non-Transitive Connectivity** Myrmic (and the original version of Chord) assumes that all (on-line) DHT nodes can communicate with each other. However, this assumption does not hold in practice. The non-transitive connectivity can be problematic in DHT because it causes inconsistency. Below we discuss how it may cause problems in Myrmic, and simple mechanisms to overcome these problems. We assume every node can communicate with the  $NA$ .<sup>6</sup> Non-Transitive connectivity may also make Nodes invisible in two cases. (1) A node  $A$  learns about  $B$  through  $NA$  (i.e.,  $A$  and  $B$  are neighbors) and can not communicate with  $B$ .  $A$  issues a update nCert request and receives the updated nCert. If  $B$  is still listed, then  $A$  knows that  $NA$  can communicate with  $B$ .  $A$  considers  $B$  as a “invisible neighbor” and stops pinging  $B$  for a pre-defined period. (2)  $A$  learns through node  $C$  about  $B$  as a proper finger, but cannot communicate with  $B$ .  $A$  marks  $B$  as a “invisible finger” but does not remove it from the finger table.  $A$  pings  $B$ ’s neighbors listed in  $B$ ’s nCert,  $B$ ’s neighbors pong with their current nCerts. If  $B$  is listed, then  $B$  is a (transient) invisible finger and should not be removed. Note  $A$  can route messages through  $B$ ’s neighbors listed in  $B$ ’s nCert, if necessary. Other negative effects including routing loops, broken return paths, and inconsistent roots either are not applicable to our protocol or can be handled with approaches similar to the ones used in [28].

## 7. RELATED WORK

Sit and Morris [30] 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 [31, 27, 32]. Lynch *et al.* [33] 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 [34, 19]. Two Sybil-resistant schemes based on social links were recently proposed in [35, 36]. 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.* [17]; 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

<sup>6</sup>Otherwise, it will be automatically removed from the ring after nCert Update protocol.

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, [17] 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 [17] 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 [37, 38] give a protocol for a “content-addressable” network that is robust to node removal. Kubiatowicz [39] 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.* [15] 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. [15] makes use of a protocol of Scheideler [22] 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 [17, 30], or routing table poisoning attack, malicious nodes conspire to fool honest nodes to include the malicious nodes into their routing tables. Singh *et al.* [18, 40] 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.* [23] mitigate eclipse attacks using induced churn. The main idea includes three components: periodically reset routing tables to constrained ones [17], limit routing table update rate, and periodically change nodes’ nodeIDs. We note that the Eclipse attack is not possible against Myrmic because we employ range verification along with 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, it is reasonable to expect adversarial interference in the future. Thus, in this paper, we introduce *Myrmic*, a novel DHT routing protocol designed to be robust against adversarial interference. We believe that *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 with a significant fraction of faulty nodes. Thus in many applications, it can be used as a drop-in, secure replacement for other existing DHT routing protocols. An important issue for future work will be the design of 1) efficient yet secure protocols against a fully adaptive adversary (who can target any node for immediate corruption), and 2) a completely distributed *NA*, since it may be impossible to define a centralized and trusted entity for some applications.

## 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. Kubiawicz, 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. Kubiawicz, "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] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *SOSP*, 2001.
- [8] A. Rowstron and P. Druschel, "Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility," in *SOSP*. ACM, 2001.
- [9] K. Park, V. S. Pai, L. L. Peterson, and Z. Wang, "Codns: Improving dns performance and reliability via cooperative lookups," in *OSDI*, 2004.
- [10] V. Ramasubramanian and E. Sirer, "The design and implementation of a next generation name service for the internet," in *SIGCOMM*, 2004.
- [11] D. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris, "Resilient overlay networks," in *ACM SOSP*, 2001.
- [12] 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.
- [13] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs," in *Crypto*, 1998.
- [14] R. J. Anderson and R. M. Needham, "Programming satan's computer," in *Computer Science Today*, 1995.
- [15] A. Fiat, J. Saia, and M. Young, "Making chord robust to byzantine attacks," in *ESA*, 2005.
- [16] 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.
- [17] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach, "Secure routing for structured peer-to-peer overlay networks," in *OSDI*, 2002.
- [18] A. Singh, M. Castro, P. Druschel, and A. Rowstron, "Defending against eclipse attacks on overlay networks," in *EWI*, 2004.
- [19] J. R. Douceur, "The sybil attack," in *Proc. of the IPTPS02*, 2002.
- [20] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *IPTPS*, 2001.
- [21] S. Crosby and D. Wallach, "Denial of service via algorithmic complexity attacks," in *USENIX Security*, 2003.
- [22] C. Scheideler, "How to spread adversarial nodes? Rotate!" in *STOC*, 2005.
- [23] T. Condie, V. Kacholia, S. Sankararaman, J. Hellerstein, and P. Maniatis, "Induced churn as shelter from routing table poisoning," in *NDSS*, 2006.
- [24] I. Osipkov, P. Wang, N. Hopper, and Y. Kim, "Robust Accounting in Decentralized P2P Storage Systems," in *ICDCS*, 2006.
- [25] OpenSSL Project Team, "Openssl," <http://www.openssl.org/>, 2006.
- [26] "Berkeley chord library," <http://i3.cs.berkeley.edu/>.
- [27] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiawicz, "Handling churn in a dht," in *USENIX Annual Technical Conference*, 2004.
- [28] M. J. Freedman, K. Lakshminarayanan, D. Rhea, and I. Stoica, "Non-transitive connectivity and dhts," in *WORLDS'05*, 2005.
- [29] 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.
- [30] E. Sit and R. Morris, "Security Considerations for Peer-to-Peer Distributed Hash Tables," in *IPTPS*, 2002.
- [31] M. Castro, M. Costa, and A. Rowstron, "Performance and dependability of structured peer-to-peer overlays," Microsoft Research, Tech. Rep. MSR-TR2003-94.
- [32] 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.
- [33] N. Lynch, D. Malkhi, and D. Ratajczak, “Atomic data access in content addressable networks,” in *IPTPS*, 2002.
  - [34] E. Friedman and P. Resnick, “The Social Cost of Cheap Pseudonyms,” *J. of Economics and Management Strategy*, 2001.
  - [35] S. Marti, P. Ganesan, and H. Garcia-Molina, “DHT Routing Using Social Links,” in *P2PDB*, 2004.
  - [36] G. Danezis, C. Lesniewski-Laas, M. F. Kaashoek, and R. Anderson, “Sybil resistant DHT routing,” in *ESORICS*, 2005.
  - [37] A. Fiat and J. Saia, “Censorship resistant peer-to-peer content addressable networks,” in *SODA*, 2002.
  - [38] J. Saia, A. Fiat, S. Gribble, A. Karlin, and S. Saroiu, “Dynamically fault-tolerant content addressable networks,” in *IPTPS*, 2002.
  - [39] K. Hildrum and J. Kubiawicz, “Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks,” in *DISC*, 2003.
  - [40] A. Singh, T.-W. J. Ngan, , P. Druschel, and D. S. Wallach, “Eclipse attacks on overlay networks: Threats and defenses,” in *Infocom*, 2006.