Secure Routing in Peer-to-Peer Distributed Hash Tables

Keith Needels
Department of Computer Science
Rochester Institute of Technology
Rochester, NY 14623
keithn@cs.rit.edu

Minseok Kwon
Department of Computer Science
Rochester Institute of Technology
Rochester, NY 14623
jmk@cs.rit.edu

ABSTRACT
Distributed hash tables (DHTs) provide efficient and scalable lookup mechanisms for locating data in peer-to-peer (p2p) networks. Several issues, however, prevent DHT-based p2p networks from being widely deployed—of which is security. Malicious peers may modify, drop, misroute lookup requests, or even collude to deny the availability of target data. To address these security concerns, we propose an extension to Chord named Sechord. The main idea is that the source can determine whether the next hop is valid or invalid by estimating how far the next hop is from its finger pointer. If the next hop is too far away from the finger pointer, especially compared to the average distance between two consecutive peers, the source can infer some ongoing malicious activities. Our modifications require no trust between two nodes except node join. Moreover, each node utilizes locally available information to evaluate hops encountered during the lookup routing process for validity. These modifications have been implemented and evaluated in the presence of malicious nodes. Our results show that Sechord significantly enhances the security of structured p2p systems at the expense of slightly increased hop count.

Categories and Subject Descriptors
C.2.2 [Network protocols]: Applications; C.2.5 [Local and wide-area networks]: Internet; D.4.4 [Communications management]: Network communication; D.4.8 [Performance]: Simulation

General Terms
Algorithms, Design, Performance

Keywords
peer-to-peer overlay networks, security, routing, distributed hash tables

1. INTRODUCTION
Since the advent of Napster in 1999 [2], peer-to-peer (p2p) systems have gained popularity as feasible file sharing applications [1]. Motivated by the success of these practical p2p systems, research efforts have been made to develop more structured p2p systems that can attain higher search efficiency and scalability [14].

While DHTs are more efficient and scalable than Napster and flooding-based systems, serious barriers exist that prevent the system from being deployed in large and open networks—one of which is security. With DHTs, we must rely on other peers to correctly forward our lookup requests to find the peer responsible for a target file. Unlike IP routing, any peer can become a malicious router and easily disrupt the entire routing. Attackers may modify, drop or misroute lookup requests. The attackers can also take responsibility for certain data to deny its availability or provide modified data to other nodes. The attackers can even forward incorrect overlay route updates.

In this paper, we explore the security concerns that DHT-based p2p systems face, and present an extension to the Chord protocol that mitigates these security concerns. Structured p2p systems using DHTs are especially vulnerable to these attacks because the nodes are organized in a special way which can easily break the entire system if some of these nodes malfunction. We design our defense mechanism targeting at three types of security threat: (i) a malicious peer does not respond to lookup requests from other peers, (ii) an attacker forwards a lookup request to some random node, not the correct next hop, and (iii) a group of attackers collude to lead lookup requests to be forwarded to another malicious node.

To remedy these concerns, we design and implement an extension to Chord named Sechord. In Sechord, the source has more control over the routing process and can further detect any suspicious activities in file search with high probability. The main idea is that the source can determine whether the next hop is valid or invalid by estimating how far the next hop is from its finger pointer. If the next hop is too far away from the finger pointer, especially compared to the average distance between two consecutive peers, the source can infer some ongoing malicious activities. Sechord also utilizes readily available information that can be easily acquired through the normal operation of Chord to evaluate possible threats. Of course, all of these modifications are executed in a fully distributed fashion. We assume that a node trusts no other nodes but the bootstrap nodes that are used for joining. Our results indicate that our protocol is able to correctly complete lookups over 90% of the time with up to half of the network consisting of compromised nodes performing naive attacks. In addition, our extension offers significant improvement over the original Chord protocol in the face of sophisticated attacks.

The rest of the paper is organized as follows. In Section 2, we discuss the security threats on structured p2p systems that we aim to address. In Section 3, we give an overview of Sechord and discuss details of its main components. In Section 4, we evaluate the
performance of Sechord using extensive simulations. In Section 5, we discuss related work. Finally, we summarize our conclusions and future work in Section 6.

2. SECURITY THREAT MODELS

We assume that attackers cannot choose their own node identifiers. As discussed in [3], each node can be prohibited from choosing its identifier if we use a certificate authority. This assumption prevents any possible attacks exploiting node identifiers. We also assume that a fraction of nodes in the system is compromised at any moment, and that some (or all) of these nodes may collude with each other. We assume that these attackers are a minority in the system—i.e., we do not consider a Sybil attack [6]. Attackers may drop p2p lookup requests, or forward requests to incorrect nodes. Attackers may also direct lookup requests to other malicious nodes possibly colluding with the attackers. Attackers may also be selective in which lookup requests they respond to correctly or not.

With these assumptions, we design and evaluate Sechord against three different types of attacks. First, a malicious node simply does not respond to lookup requests from other nodes (dropping lookup requests). Second, an attacker forwards a lookup request to some random next hop preventing the lookup request from reaching the final destination. This type of attack is more difficult to defend against than the first attack since the attacker may appear to be cooperative by providing the next hop (misrouting lookup requests). Third, a group of attackers collude to lead lookup requests to be forwarded to a malicious node. The attackers maintain two finger tables: 1) the legitimate finger table that the attackers use for routing among themselves, and 2) the finger table that consists of the first succeeding malicious node for each node at the legitimate finger table. An attacker forwards and circulates a lookup request through malicious nodes, so that the lookup request is likely to end up at a wrong place (colluding for subring attacks).

3. SECURE ROUTING IN SECHORD

3.1 Design Overview

To defend against the attacks previously described, we propose to utilize locally available statistical data, specifically the average numerical distance between consecutive node identifiers. Our objective is to detect any suspicious routing anomaly during the routing process of a lookup request and recover from that.

As a lookup request is routed towards the destination, we determine whether a next hop is valid or invalid. One important property in Chord is that \(|p-f| \leq |p-q| \) and \(|q-f| \leq |p-q|\) where \(p\) and \(q\) are two consecutive peers, \(f\) is a finger pointer falls somewhere between \(p\) and \(q\), and \(|x-y|\) is the numerical distance between the identifiers of peers \(x\) and \(y\). Based on this property, we compare the distance between a finger pointer and the actual node identifier referred by this finger pointer with the average distance between two consecutive peers computed from the finger table at the source. If these two distances are significantly far off, we suspect a misbehavior in the routing. In that case, we backtrack to the previous node on the route and request a next hop that makes less progress towards the destination in an attempt to avoid the node that has provided us with the invalid hop.

To this end, we slightly modify the original Chord protocol. We store the identifiers of the successor and predecessor of each node in the finger table. This extra information helps us compute the average numerical distance between two node identifiers. We also modify the Chord routing algorithm using iterative routing as opposed to recursive routing. In Chord, a lookup request initiated by the source is first passed to the next hop, and then the next hop forwards the request to the third hop recursively. With this approach, the source has no control over the routes its lookup requests take. In contrast, the iterative routing allows the source to control the entire routing steps. The source itself contacts each individual node on the route and requests the next hop as the lookup request travels towards the destination.

3.2 Hop Verification

In Chord, the finger table at node \(id\) contains \(m\) entries, and the finger pointer \(f_i\) for entry \(i\) in the table is \(f_i = (id + 2^i) \mod 2^m\), where \(m\) is the bit length of the identifiers used in the network and \(0 \leq i < m\). More precisely, entry \(i\) in the table points to the first node \(n\) that is the successor of \(f_i\) clockwise on a circle of numbers from 0 to \(2^m - 1\). (We also refer to a node like \(n\) as a finger table entry throughout the paper). \(f_i\) indeed falls between \(p\) and \(n\) where \(p\) is the predecessor of \(n\) on the circle. It thus follows that \(|f_i - n| \leq E(id)|\) where \(E(id)|\) is the average distance between a finger table entry and its predecessor. We accept \(n\) if \(|f_i - n| \leq E(id)| + \alpha \times \sigma\); we reject \(n\) otherwise. Here \(\sigma\) is the standard deviation of these distance samples and \(\alpha\) is the parameter that controls the weight of the standard deviation over the average. We can control the rates of false positives and false negatives by changing the value of \(\sigma\).

This hop distance test strictly constraints the scope of the next hop identifier. Without control over the nodes in that scope, an attacker cannot easily fool the source into using another attacking node as the next hop. Since nodes cannot arbitrarily place themselves wherever they wish, it becomes much more difficult for a malicious node to forward a lookup request from the source to a node other than the correct next node on the route.

3.3 Backtracking

In Chord routing, each peer selects the peer that most closely precedes the target identifier as the next hop. When a faulty node is detected in Sechord, however, we retreat to the previous node on the route and attempt its next closest preceding node to the destination in the finger table. This approach, of course, offers less progress, but allows us to route around the faulty node. If all of the preceding nodes turn out to be faulty in the node, we retreat even further to the previous node of this node and repeat the process. All faulty nodes or whose finger pointers are all faulty are maintained in a temporary black list, so that these peers will not be tried again for the duration of this routing attempt. To prevent the need to query the same node multiple times, we also request the entire finger table of a node the first time the node appears on the path and cache the table for the duration of the lookup attempt.

Every lookup request in Chord is delivered to the node that immediately precedes the target. If this immediate preceding node is faulty, however, our backtracking scheme does not help much in finding another route around this faulty node since all of the routes must pass through the immediate preceding node. To address this problem, we seek to find a non-faulty preceding node that contains the destination in its finger table, and directly jump to the destination bypassing all of the intermediate faulty nodes. To accomplish this, each node is made aware of the predecessor of every node in the finger table. When we find a non-faulty node with the destination in its finger table, we ensure that the target key identifier falls between the destination and its predecessor. In this case, however, we do not immediately bypass the rest of the nodes on a route, but instead bypass only when the non-faulty node has no more available hops that we can attempt. This is because nodes further away from the destination are more likely to have
outdated successor information due to the possible delay with the \texttt{fix\_fingers()} method.

### 3.4 Routing

Our routing algorithm is a modified version of \texttt{find\_successor()} in Chord, and utilizes the hop verification and backtracking algorithms. We keep track of discovered nodes in the routing stack. Each entry in the stack contains a tuple of \(<\text{id, fingertable, index}>\) that denotes the identifier, finger table, and current backtracking index of that particular node. We also maintain a blacklist containing the nodes that have failed the hop verification process or that have no more nodes in their finger table to compute the next hop. We also limit the total number of hops below the specified maximum.

The stack initially contains only one node that performs the lookup. The routing loop continues as long as routers are available in the stack and the maximum number of hops is not exceeded. During each iteration, the router at the top of the stack is popped off, its backtrack index is incremented, and the lookup is performed with this incremented backtrack information. This means that when a router is popped off for the \(i\)th time, its backtrack index becomes \(i\) and the backtracking algorithm uses the \(i\)th closest preceding node in the router’s finger table. We then assure that the returned next hop is valid by the hop verification and the next hop is not already included in the stack.

After these preliminary tests, we examine three conditions as follows. First, if the next hop succeeds the target identifier, we return it as the destination. Second, if the next hop is on the black list, we simply push the router popped off back to the stack. This router will be popped off again later with an increased backtrack index value (a prior finger table entry). Third, if the next hop is neither the destination nor on the black list, we contact the node and request its finger table – this is indeed a legitimate next hop. We then add this node to stack with an initial backtrack index of 0.

Figure 1 shows an example of the routing algorithm in Sechord. The white nodes are compromised while the dark nodes are not. The routing loop continues as long as routers are available in the stack, and the maximum number of hops is not exceeded. During each iteration, the router at the top of the stack is popped off, its backtrack index is incremented, and the lookup is performed with this incremented backtrack information. This means that when a router is popped off for the \(i\)th time, its backtrack index becomes \(i\) and the backtracking algorithm uses the \(i\)th closest preceding node in the router’s finger table. We then assure that the returned next hop is valid by the hop verification and the next hop is not already included in the stack.

After these preliminary tests, we examine three conditions as follows. First, if the next hop succeeds the target identifier, we return it as the destination. Second, if the next hop is on the black list, we simply push the router popped off back to the stack. This router will be popped off again later with an increased backtrack index value (a prior finger table entry). Third, if the next hop is neither the destination nor on the black list, we contact the node and request its finger table – this is indeed a legitimate next hop. We then add this node to stack with an initial backtrack index of 0.

### 3.5 Statistics, Node Join, and Maintenance

Each entry in the finger table contains the index, node identifier, node remote reference (same as Chord), plus additionally the predecessor identifier and successor list identifiers. These two additional fields allow a node to generate samples and compute the average distance between nodes as well as the standard deviation. All of these fields are obtained when the \texttt{fix\_fingers()} method in Chord is invoked to update finger tables (see [14] for details). It should be noted that a node may lie about the identifiers of its predecessor and successors. To address this issue, we can first make sure that the identifiers provided correspond to valid nodes by requesting full certificates instead of only identifiers.

For a new node to securely join, we assume uncompromised bootstrap servers that must be found out of band. We also assume that the new node and the bootstrap servers trust each other to prevent the new node from starting at a malicious node. Note that we assume trust between two nodes only in this case. The bootstrap servers then perform a lookup for each finger pointer identifier at the new node to construct the initial finger table at the new node.

We also need to slightly modify \texttt{fix\_fingers()} in Chord that is invoked when maintaining the finger tables. If we receive a new finger entry closer than the old one, we accept it; otherwise (a new finger entry further away than the old one), we compare the new entry with the old one as well as the successor list to assure that all of the nodes with preceding identifiers to the new node’s identifier have indeed left the network. If none of these nodes are in the network, we accept the update; otherwise, we reject it, and use the closest succeeding node found in the current finger table.

### 4. PERFORMANCE EVALUATION

We have implemented our Sechord protocol described above and simulated it to quantify its advantages over Chord. Our implementation is written in Java and requires Java version 1.5 or later to compile and execute. The simulation runs iteratively, not multi-threaded; the iterative approach is simple and allows us to examine most of the functionalities in the implementation without any problem.

We use the following performance metrics for evaluating Sechord: 1) the effects of malicious nodes that drop lookup requests, 2) the effects of malicious nodes that route lookup requests to other random malicious nodes instead of the correct next hop, and 3) the vulnerability of the system when malicious nodes form a subring of themselves and forward all received lookup requests within the subring.

We use the term lookup success ratio to denote the number of successful lookups and total the number of lookups attempted. We also measure average hop count that denotes the average of all the hop counts consumed in the lookups.

### 4.1 Experimental Setup and Results

Chord usually deals with unresponsive nodes (mainly due to failure) through the \texttt{stabilize()} and \texttt{check\_predecessor()} functions (see [14] for details). These malicious nodes can, however, bypass those two functions to remain in the system and yet do not respond to lookup requests. Our algorithm can deal with such attacks as well as any failed nodes that may not have been cleaned up. We experiment with a network of 1000 nodes in which 0-500 nodes are malicious. For each different set of malicious nodes, we create 100 different networks and perform 1000 random lookups. In our networks, we assume that for every two peers, there exists an overlay link. We set the standard deviation to \(\infty\) to disable the hop verification, so that we can examine how well our algorithm copes with these unresponsive nodes independent of the hop verification.

Figure 2 illustrates the percentage of successful lookups among
all the lookups for both Chord and Sechord. We find that Sechord attains much higher percentages of successful lookups than Chord when 0-50% of nodes drop all lookup requests. Unlike Chord, Sechord can route around compromised nodes and find the node responsible for a keyword approximately 95% of the time. In contrast, the percentage of successful lookups in Chord decreases sharply as the percentage of compromised nodes increases. The lookup success ratio decreases to around 75% with 5% compromised nodes, 27% lookup fails if at least one node on the path to the destination drops the lookup request. Since the average path length from source to destination is 4.8 hops in our simulations, the probability that a lookup request arrives at the destination is \( f^{1.8} \), where \( f \) is the fraction of uncompromised nodes. As \( f \) decreases, the probability drops rapidly.

The higher percentages of successful lookups in Sechord, however, come at the expense of the increased number of hops. Figure 3 indicates that the average hop count increases as more nodes drop lookup requests, and doubles when 28% of nodes drop the requests. The average hop count becomes almost four times higher than the average hop count in Chord (4.8) when 50% of nodes are compromised.

To examine the hop verification algorithm, we create an attack in which a malicious node directs lookup requests to other random malicious nodes. This attack bounces the lookup requests all over the network while never reaching the destination. We use the same setup as the previous experiment except the standard deviation (3.0) and the pruning parameter (1.0).

In Figure 4, the result indicates that the lookup success ratio of Sechord is higher than that of Chord when this random routing threat is present. The lookup success ratio of Chord decreases rapidly as more nodes are involved with random misrouting. When 25% nodes are compromised, the success ratio is reduced to 28%; with 50% compromised nodes, the success ratio becomes no higher than 10%. In contrast, the success ratio of Sechord stays no lower than 90% for all percentages of compromised nodes. The hop verification algorithm in Sechord helps detect malicious hops when the lookup request is forwarded to a random location. The average hop count with random routing is similar to the previous case in Figure 3.

A more vicious attack is possible when a group of malicious nodes collude and form a subring in which each node maintains an extra finger table. In this finger table, each entry contains the first malicious node that succeeds each finger pointer. If a lookup query reaches one of these malicious nodes, the query is captured by the colluding subring and may not be able to be out of this subring ever. The goal of the attacker is to emulate correct behavior as closely as possible while still controlling the lookup process. This type of attack is hard to detect even via our hop verification algorithm because the nodes used in the finger table are the closest malicious nodes to the correct entry. We use the same network configuration except the standard deviation that are set to 1.75 and 3.0.

The lookup success ratio depicted in Figure 5 shows that Sechord outperforms Chord even though its ratio is not as high as the previous two cases. For example, with 20% of the nodes compromised, only 48.6% of lookups succeed using Chord, while 86% succeed with Sechord when the standard deviation is 1.75. It should be
noted that as the fraction of colluding nodes increases, both the protocols ultimately see their success rate improved. When 40% of nodes are compromised, even if we were fooled by the attackers every time, 40% of our lookups would still return the correct destinations because 40% of the correct destinations are malicious nodes. The success ratio thus never becomes lower than the percent of colluding nodes as the lower bounds.

The average hop count is illustrated in Figure 6. The average hop count of Sechord is overall higher than Chord, and gradually increase as the number of colluding nodes increases. The average hop count when the standard deviation is 1.75 are higher than those with a standard deviation of 3.0. Note that, however, the lookup success ratio with the standard deviation of 1.75 is greater than the success ratio with a standard deviation of 3.0. This tradeoff is due to the balance between false positives and false negatives.

5. RELATED WORK

Castro et al. [3] were one of the early studies that investigated security issues as well as their solutions on routing for structured p2p overlay networks. They discuss attacks on node ID assignment, routing table maintenance, and message forwarding. The solutions include certified node IDs, constrained routing tables, routing failure tests, and redundant routing among many others. Our hop verification method adopts the idea of routing failure tests. Our approach, however, verifies routing hops instead of full routes. This allows us to react and route around misbehaving nodes as they are detected rather than reacting to a bad route by flooding lookup requests through our neighbors. Sit and Morris [12] also discuss security problems in p2p systems based on distributed hash tables.

The Sybil attack [6] arises when a single attacker joins a p2p network with numerous identities so that the attacker can control a large portion of the network. If an attacker compromises a large fraction of the network, the attacker can also eclipse correct nodes and disrupt correct operations. This sort of attack is called the Eclipse attack which is considered more general than the Sybil attack [10]. There are also a few other papers that aim to address the Sybil attack to some extent using distributed solutions [5].

Adversaries can also attempt to intercept traffic and fool other nodes to use compromised nodes as neighbors. These routing-table poisoning attacks can be addressed using induced churn that utilizes periodic routing-table resets, unpredictable identifier changes, and a rate limit on routing-table updates [4]. Fireflies [7] is a scalable protocol for supporting intrusion-tolerant network p2p overlays. Maniatis et al. [8] have investigated a spectrum of denial-of-service attacks from low-bit to high-bit rates on p2p systems.

TrustGuard [13] is a safeguard framework that provides a highly dependable and yet efficient reputation system. The framework aims at countering detrimental vulnerabilities in reputation systems. Agyaat [11] is a decentralized p2p system that enables privacy-preserving mutual anonymity without sacrificing the performance benefits in lookups. Additionally, a variety of security attacks and issues are discussed in [15, 9].

6. CONCLUSIONS

In p2p systems, misbehaving nodes may instigate severe disruptions even when they only exist in small numbers. We have presented a mechanism named Sechord that helps mitigate such security concerns in structured p2p systems that use distributed hash tables (DHTs). In particular, Sechord intends to defend against attacks that aim at disrupting their routing and forwarding. Sechord utilizes Chord as its underlying architecture, but modifies the routing algorithm so that the source has more control over routing decisions. The source can ensure the integrity of intermediate hops by comparing the returned hop distances with the average hop distance. Sechord requires only slight modifications to Chord; the verification method also makes use of information that is readily and locally available to each node; and the nodes are not required to trust each other. Due to this simplicity and generality, our approach can be adopted to other DHT-based p2p systems easily. Our results show that Sechord easily deals with simple attacks such as dropped lookup requests and randomly misrouted lookup requests. The results further indicate that Sechord improves the lookup success ratio even with more sophisticated attacks like a subring of colluded attackers in comparison with Chord, especially when less than 20% of nodes are compromised.

7. REFERENCES