Security Considerations for Peer-to-Peer Distributed Hash Tables

Emil Sit and Robert Morris

Laboratory for Computer Science, MIT
200 Technology Square, Cambridge, MA 02139, USA
{sit,rtm}@lcs.mit.edu

Abstract. Recent peer-to-peer research has focused on providing efficient hash lookup systems that can be used to build more complex systems. These systems have good properties when their algorithms are executed correctly but have not generally considered how to handle misbehaving nodes. This paper looks at what sorts of security problems are inherent in large peer-to-peer systems based on distributed hash lookup systems. We examine the types of problems that such systems might face, drawing examples from existing systems, and propose some design principles for detecting and preventing these problems.

1 Introduction

Peer-to-peer systems present an interesting security problem as there is no central system to protect. Instead, the nodes must work together to ensure correct and secure behavior. Unfortunately, deployment on an open network, such as the Internet, implies that there will be malicious nodes in the system. These nodes will try and disrupt the system or subvert it to their advantage. Peer-to-peer systems must be designed to operate correctly even in these situations.

A number of recent systems are built on top of peer-to-peer distributed hash lookup systems [5, 6, 9, 10]. Lookups for keys are performed by routing queries through a series of nodes; each of these nodes uses a local routing table to forward the query towards the node that is ultimately responsible for the key. These systems can be used to store data, for example, as a distributed hash table or file system [1, 7]. Other projects take advantage of other aspects of the lookup system, such as the properties of lookup routing [8]. In this paper, we will examine security concerns that are particular to distributed hash tables.

One class of attacks on distributed hash tables causes the system to return incorrect data to the application. Fortunately, the correctness and authenticity of data can be addressed using cryptographic techniques such as self-certifying path names [3]. These techniques allow the system to detect and ignore inauthentic data.

This paper focuses on the remaining attacks — those that threaten the liveness of the system by preventing participants from finding data. We begin by presenting a common framework for distributed hash tables in Section 2, followed by the basic adversary model in Section 3. The core of the paper is in

Table 1. Design Principles

Define verifiable system invariants (and verify them!)
Allow the querier to observe lookup progress.
Assign keys to nodes in a verifiable way.
Server selection in routing may be abused.
Cross-check routing tables using random queries.
Avoid single points of responsibility.

Section 4, where we present a series of examples of particular weaknesses in existing distributed hash lookup systems. From these discussions, we derive a set of general design principles, shown in Table 1. We conclude in Section 5.

2 Background

Typical distributed hash tables consist of a storage API layered on top of a lookup protocol. Lookup protocols share a few basic components:

1. a key identifier space,
2. a node identifier space,
3. rules for associating keys to particular nodes,
4. per-node routing tables that refer to other nodes, and
5. rules for updating routing tables as nodes join and fail.

The lookup protocol maps a desired key identifier to the IP address of the node responsible for that key. A storage protocol layered on top of the lookup protocol then takes care of storing, replicating, caching, retrieving, and authenticating the data. CAN [5], Chord [9] and Pastry [6] all fit into this general framework.

Routing in the lookup is handled by defining a distance function on the identifier space so that distance can be measured between the current node and the desired key; the responsible node is defined to be the node closest to the key.

Lookup protocols typically have an invariant that must be maintained in order to guarantee that data can be found. For example, the Chord system arranges nodes in a one-dimensional (but circular) identifier space; the required invariant is that every node knows the node that immediately follows it in the identifier space. If an attacker could break this invariant, Chord would not be able to look up keys correctly.

Similarly, the storage layer will also maintain some invariants in order to be sure that each piece of data is available. In the case of DHash [1], a storage API layered on Chord used by CFS, there are two important invariants. First, it must ensure that the node that Chord believes is responsible for a key actually stores the data associated with that key. Since nodes can fail, it is also important that DHash maintain replicas of each piece of data, and that those replicas be at predictable nodes. An attacker could potentially target either of these invariants.

3 Adversary Model

The adversaries that we consider in this paper are participants in a distributed hash lookup system that do not follow the protocol correctly. Instead, they seek to mislead legitimate nodes by providing them with false information.

We assume that a malicious node is able to generate packets with arbitrary contents (including forged source IP addresses), but that a node is only able to examine packets addressed to itself. That is, malicious nodes are *not* able to overhear or modify communication between other nodes. The fact that a malicious node can only receive packets addressed to its own IP address means that an IP address can be used as a weak form of node identity; if a node receives a packet from an IP address, it can verify that the packet's sender owns the address by sending a request for confirmation to that address. We also consider malicious nodes that conspire together, but where each one is limited as above. This allows an adversary to gather additional data and act more deviously by providing false but "confirmable" information.

The rest of the paper will examine the ways in which malicious nodes can use these abilities to subvert the system.

4 Attacks and Defenses

This section is organized into attacks against the routing, attacks against the data storage system, and finally some general considerations.

The first line of defense for any attack is detection. Many attacks can be detected by the node being attacked because they involve violating invariants or procedure contracts. However, it is less clear what to do once an attack has been detected. A node may genuinely be malicious, or it may have failed to detect that it was being tricked. Thus, our discussion focuses on methods to detect and possibly correct inconsistent information. We will see that achieving *verifiability* underlies all of our detection techniques.

4.1 Routing Attacks

The routing portion of a lookup protocol involves maintaining routing tables and then dispatching requests to the nodes in the routing table. It is critical that routing is correct in a distributed hash table. However, there is considerable room for an adversary to play in existing systems. These attacks can be detected if the system *defines verifiable system invariants (and verifies them).* When invariants fail, the system must have a recovery mechanism.

Incorrect Lookup Routing. An individual malicious node could forward lookups to an incorrect or non-existent node. Since the malicious node would be participating in the routing update system in a normal way, it would appear to be alive, and would not ordinarily be removed from the routing tables of other nodes. Thus re-transmissions of the misdirected lookups would also be sent to the malicious node.

Fortunately, blatantly incorrect forwarding can often be easily detected. At each hop, the querier knows that the lookup is supposed to get "closer" to the key identifier. For example, in Pastry, each hop should match the key identifier in at least one more digit than the last. The querier should check for this so that this attack can be detected. If such an attack is detected, the querier might recover by backtracking to the last good hop and asking for an alternate step.

In order for the querier to be able to perform this check, however, each step of progress must be visible to the querier. For example, CAN proposes an optimization where each node keeps track of the network RTTs to neighbor nodes and forwards to the neighbor with the best ratio of progress to RTT. This implies that queries are generally forwarded without interacting with the querier. Thus in CAN, a querier simply can not verify forward progress. One should *allow the querier to observe lookup progress.*

A malicious node might also simply declare (incorrectly) that a random node is the node responsible for a key. Since the querying node might be far away in the identifier space, it might not know that this node is, in fact, not the closest node. This could cause a key to be stored on an incorrect node or prevent the key from being found. This can be fixed with two steps.

First, the querier should ensure that the destination itself agrees that it is a correct termination point for the query. In Chord, the predecessor returns the address of the query endpoint (the "successor") instead of the endpoint itself, making this attack possible — a malicious node can cause the query to overshoot the correct successor. Since the querier Q does not know about the true successor S, a malicious predecessor P could forward to some node $S' > S$. This can cause DHash to violate its storage location invariant. However, if S' is good, then it can see that it should not be responsible for this key and can raise an error.

Second, the system should *assign keys to nodes in a verifiable way.* In particular, in some systems, keys are assigned to the node that is closest to them in the identifier space. Thus in order to assign keys to nodes verifiably, it is sufficient to derive node identifiers in a verifiable way. Contrast this to CAN, which allows any node to specify its own identity. This makes it impossible for another node to verify that a node is validly claiming responsibility for a key. Some systems, like Chord, make an effort to defend against this by basing a node's identifier on a cryptographic hash of its IP address and port.[1] Since this is needed to contact the node, it is easy to tell if one is speaking to the correct node.

Systems may want to consider deriving long-term identities based on public keys. This has performance penalties due to the cost of signatures, but would allow systems to have faith on the origin of messages and the validity of their contents. That is, public keys would facilitate the verifiability of the system. For example, a certificate with a node's public key and address can be used by new nodes to safely join the system.

[1] The hash actually also includes a virtual node identifier, which will be relevant in Section 4.2

Incorrect Routing Updates. Since each node in a lookup system builds its routing table by consulting other nodes, a malicious node could corrupt the routing tables of other nodes by sending them incorrect updates. The effect of these updates would be to cause innocent nodes to misdirect queries to inappropriate nodes, or to non-existent nodes. However, if the system knows that correct routing updates have certain requirements, these can be verified. For example, Pastry updates require that each table entry has a correct prefix. Blatantly incorrect updates can be easily identified and dropped. Other updates should only be incorporated into a node's routing table after it has verified itself that the remote node is reachable.

A more subtle attack would be to take advantage of systems that allow nodes to choose between multiple correct routing entries. For example, CAN's RTT optimization allows precisely this in order to minimize latency. A malicious node can abuse this flexibility and provide nodes that are undesirable. For example, it might choose an unreliable node, one with high latency, or even a fellow malicious node. While this may not affect strict correctness of the protocol, it may impact applications that may wish to use the underlying lookup system to find nodes satisfying certain criteria. For example, the Tarzan anonymizing network [2] proposes the use of Chord as a way of discovering random nodes to use in dynamic anonymizing tunnels. Any flexibility in Chord might allow an adversary to bias the nodes chosen, compromising the design goals of Tarzan. Applications should be aware that *server selection in routing may be abused.*

Partition. In order to bootstrap, a new node participating in any lookup system must contact some existing node. At this time, it is vulnerable to being partitioned into an incorrect network. Suppose a set of malicious nodes has formed a parallel network, running the same protocols as the real, legitimate network. This parallel network is entirely internally consistent and may even contain some of the data from the real network. A new node may join this network accidentally and thus fail to achieve correct results. One of the malicious nodes might also be cross-registered in the legitimate network and may be able to cause new participants to be connected to the parallel network even if they have a valid bootstrap node.

Partitions can be used by malicious nodes to deny service or to learn about the behavior of clients that it would otherwise be unable to observe. For example, if a service was made available to publish documents anonymously, an adversary could establish a malicious system that shadows the real one but allows it to track clients who are reading and storing files.

In order to prevent a new node from being diverted into an incorrect network, it must bootstrap via some sort of trusted source. This source will likely be out-of-band to the system itself. When rejoining the system, a node can either use these trusted nodes, or it can use one of the other nodes it has previously discovered in the network. However, building trust metrics for particular nodes can be dangerous in a network with highly transient nodes that lack any strong sense of identity. If a particular address is assigned via DHCP, for example, a node could be malicious one day but benign the next. Again, use of public keys may reduce this risk.

If a node believes it has successfully bootstrapped in the past, then it can detect *new* malicious partitions by cross-checking results. A node can maintain a set of other nodes that it has used successfully in the past. Then, it can *cross-check routing tables using random queries.*[2] By asking those nodes to do random queries and comparing their results with its own results, a node can verify whether its view of the network is consistent with those other nodes. Note that randomness is important so that a malicious partition can not distinguish verification probes from a legitimate query that it would like to divert. Conversely, a node that has been trapped in a malicious partition might accidentally discover the correct network in this manner, where the "correct" network here is defined as the one which serves desired data.

4.2 Storage and Retrieval Attacks

A malicious node could join and participate in the lookup protocol correctly, but deny the existence of data it was responsible for. Similarly, it might claim to actually store data when asked, but then refuse to serve it to clients. In order to handle this attack, the storage layer must implement replication. Replication must be handled in a way so that no single node is responsible for replication or facilitating access to the replicas; that node would be a single point of failure. So, for example, clients must be able to independently determine the correct nodes to contact for replicas. This would allow them to verify that data is truly unavailable with all replica sites. Similarly, all nodes holding replicas must ensure that the replication invariant (e.g. at least r copies exist at all times) is maintained. Otherwise, a single node would be able to prevent all replication from happening. In summary, *avoid single points of responsibility.*

Clients doing lookups must be prepared for the possibility of malicious nodes as well. Thus, they must consult at least two replica sites in order to be sure that either all of the replicas are bad or that the data is truly missing.

As an example, DHash does not follow this principle: only the node immediately associated with the key is responsible for replication. However, even if the storing node performed replication, DHash would still be vulnerable to the actual successor lying about the r later successors. Replication with multiple hash functions, as proposed in CAN, is one way to avoid this reliance on a single machine.

This attack can be further refined in a system that does not assign nodes verifiable identifiers. In such a system, a node can choose to become responsible for data that it wishes to hide. DHash continues to be at risk here, despite Chord having verifiable node identifiers, because the identifier is derived from a hash of the node's IP address, port number and virtual node number. Since a person in control of a single node can run a large number of virtual nodes, they can still effect some degree of choice in what data they wish to hide. IPv6 or sparsely used IPv4 networks may also allow a single host to appear to run many nodes.

[2] Of course, without a sense of node identity that is stronger than IP address, this is still dangerous.

4.3 Miscellaneous Attacks

Inconsistent Behavior. Any of the attacks here can be made more difficult to detect if a malicious node presents a good face to part of the network. That is, a malicious node may choose to maximize its impact by ensuring that it behaves correctly for certain nodes. One possible target would be nodes near it in the identifier space. These nodes will not see any reason to remove the node from their routing tables despite the fact that nodes that are distant see poor or invalid behavior. This may not be a serious problem if queries must be routed through close nodes before reaching the target node. However, most routing systems have ways of jumping to distant points in the identifier space in order to speed up queries.

Ideally, distant nodes would be able to convince local nodes that the "locally good" malicious node is in fact malicious. However, without public keys and digital signatures, it is not possible to distinguish a report of a "locally good" node being malicious, from a malicous report trying to tarnish a node that is actually benign. On the other hand, with public keys, this can be proven by requiring nodes to sign all of their responses. Then a report would contain the incorrect response and the incongruity could be verified. Lacking this, each node must make its own determination as to whether another node is malicious.

Overload of Targeted Nodes. Since an adversary can generate packets, it can attempt to overload targetted nodes with garbage packets. This is a standard denial of service attack and not really a subversion of the system. This will cause the node to appear to fail and the system will be able to adapt to this as if the node had failed in some normal manner. A system must use some degree of data replication to handle even the normal case of node failure. This attack may be effective if the replication is weak (i.e. the malicious nodes can target all replicas easily) or if the malicious node is one of the replicas or colluding with some of the replicas.

The impact of denial of service attacks can be partially mitigated by ensuring that the node identifier assignment algorithm assigns identifiers to nodes randomly with respect to network topology. Additionally, replicas should be located in physically disparate locations. These would prevent a localized attack from preventing access to an entire portion of the key space. If an adversary did wish to shut out an entire portion of the key space, it would have to flood packets all over the Internet.

Rapid Joins and Leaves. As nodes join and leave the system, the rules for associating keys to nodes imply that new nodes must obtain data (from replicas) that was stored by nodes that have left the system. This rebalancing is required in order for the lookup procedures to work correctly. A malicious node could trick the system into rebalancing unnecessarily causing excess data transfers and control traffic. This will reduce the efficiency and performance of the system; it may even be possible to overload network segments. This attack would work best if the attacker could avoid being involved in data movement since that would consume the bulk of the bandwidth. An adversary might try to convince the system that a particular node was unavailable or that a new node had (falsely) joined. However, our model allows the adversary no (low-bandwidth) way of accomplishing the former; the latter case presumably will involve acknowledged data transfers that the adversary can not correctly acknowledge. Any other rebalancing would involve the adversary node itself, requiring it to be involved in the data movement.

Note that any distributed hash table must provide a mechanism for dealing with this problem, regardless of whether there are malicious nodes present. Early studies have shown that in some file sharing systems, peers join and leave the system very rapidly [4]. The rate of replication and amount of data stored at each node must be kept at levels that allow for timely replication without causing network overload when even regular nodes join and leave the network.

Unsolicited Messages. A malicious node may be able to engineer a situation where it can send an unsolicited response to a query. For example, consider an iterative lookup process where querier Q is referred by node E to node A. Node E knows that Q will next contact A, presumably with a follow-up to the query just processed by E. Thus, E can attempt to forge a message from A to Q with incorrect results.

The best defense against this would be to employ standard authentication techniques such as digital signatures or message authentication codes. However, digital signatures are currently expensive and MACs require shared keys. A more reasonable defense may be to include a random nonce with each query and have the remote end echo the nonce in its reply. This would essentially ensure the origin of the response.

5 Conclusion

This paper presents a categorization of the basic attacks that peer-to-peer hash lookup systems should be aware of. It discusses details of those attacks as applied to some specific systems, and suggests defenses in many cases. It abstracts these defenses into this set of general design principles: 1) define verifiable system invariants, and verify them; 2) allow the querier to observe lookup progress; 3) assign keys to nodes in a verifiable way; 4) be wary of server selection in routing; 5) cross-check routing tables using random queries; and 6) avoid single points of responsibility.

References

[1] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM SOSP* (Banff, Canada, Oct. 2001), pp. 202–215.

[2] Freedman, M. J., Sit, E., Cates, J., and Morris, R. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the First International Workshop on Peer-to-Peer Systems* (Cambridge, MA, Mar. 2002).

[3] Fu, K., Kaashoek, M. F., and Mazières, D. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2000), pp. 181–196.

[4] Krishnamurthy, B., Wang, J., and Xie, Y. Early measurements of a cluster-based architecture for P2P systems. In *Proceedings of the First ACM SIGCOMM Internet Measurement Workshop* (San Francisco, California, Nov. 2001), pp. 105–109.

[5] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM* (San Diego, California, Aug. 2001), pp. 161–172.

[6] Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)* (Nov. 2001).

[7] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM SOSP* (Banff, Canada, Oct. 2001), pp. 188–201.

[8] Rowstron, A., Kermarrec, A.-M., Castro, M., and Druschel, P. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication: Third International COST264 Workshop* (Nov. 2001), J. Crowcroft and M. Hofmann, Eds., vol. 2233 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 30–43.

[9] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM* (San Diego, California, Aug. 2001), pp. 149–160.

[10] Zhao, B., Kubiatowicz, J., and Joseph, A. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, Apr. 2001.