Exploiting Trust in Peer-to-Peer Systems

Srinivas Visvanathan and John H. Hartman
{srini, jhh}@cs.arizona.edu

Technical Report: TR03-09

December 1, 2003

Abstract

Second generation peer-to-peer systems [5, 6] have shown a lot of promise with many desirable properties such as scalability, self-configuration, automatic load balancing etc. However the open nature of these systems also makes them vulnerable to Byzantine attacks.

These systems are designed to be fault tolerant in the presence of a few malicious nodes. We however believe that even a handful of evil nodes could disrupt the system by exploiting certain weaknesses that we have identified. In these systems, a single maclicious node, can present multiple identities to the system. This allows them to emulate multiple nodes simultaneously and also allows them limited control of their placement in overlay networks[4, 7] that are integral to the system. We devised attacks based on these weaknesses to disrupt lookup and storage operations in the peer-to-peer storage systems, CFS[5] and PAST[6]. We show that it is possible to exploit these weaknesses to attack these systems and have evaluated the feasibility of these attacks.

1 Introduction

Peer-to-peer systems are a hot topic in operating systems research. Though the area initially became popular with the advent of peer-to-peer file sharing applications like Napster [1], Gnutella [3] and Freenet [2], systems researchers were quick to identify many desirable properties of peer-to-peer systems, in general, such as scalability and symmetry.

The first generation peer-to-peer systems were developed for commercial [1] or political reasons [2, 3]. Their basic design did not make them very efficient; Napster had a centralized directory, Gnutella used a broadcast-based protocol and Freenet used probabilistic routing that sacrificed efficiency for anonymity. Many researchers were quick to see the opportunity for improvement and this led to the development of several second generation peer-to-peer systems [5, 6]. All these systems organized servers in the Internet as an overlay network that provided routing and configuration services. Different kinds of peer-to-peer applications could be built on top of them. These systems were more efficient and could provide guarantees about reliability and availability. They were also designed to tolerate fail-stop failures (node failure that can easily be detected by other nodes). However, researchers had not thoroughly addressed the issue of Byzantine failures in these systems (failures where a node appears active, but behaves incorrectly, or more likely, maliciously). The basic openness of peer-to-peer systems seem to make them especially vulnerable to Byzantine failures.

The aim of this project is to examine the issue of trust in peer-to-peer systems and see how it could be misused by malicious hosts. We wanted to do this by studying a prototype peer-to-peer system and seeing how we could disrupt its operations. For this project we chose to work with a prototype peer-to-peer system called Chord [4] that was developed at MIT and is used in CFS [5]. We were successful in identifying vulnerabilities in the system related to the issue of trust. We were also able to modify Chord nodes to operate in a malicious manner and disrupt the operations of the system. We have also investigated how applicable these issues and attacks are in the Pastry [7] system that is used in PAST [6].

In the next section we provide some background about the design and implementation of second generation peer-to-peer systems. Section 3 describes the weaknesses we were able to identify and section 4 describes how we attempted to exploit these weaknesses. In section 5 we review work done by other researchers and conclude in section 6.

2 Background

Second generation peer-to-peer systems are organized into two distinct layers. The lower layer is made up of Internet servers that organize themselves into an overlay network. This layer basically provides a consistent, distributed hashing scheme. Nodes in this network are assigned identifiers or ID's. Keys representing resources are also assigned ID's drawn from the same space. The network of nodes implement a hashing scheme that maps resource keys onto node keys i.e. each resource is assigned to a node that is responsible for it. The hashing scheme is distributed because the information required to perform a mapping is distributed across several nodes. It is consistent which means that when nodes join or leave the system, the mapping is updated to reflect the modified set of nodes in the system. The number of nodes that have to update their mapping information, due to the change in system state, should ideally be minimal.

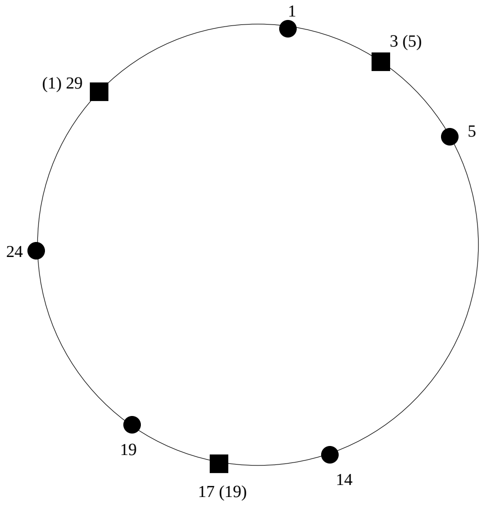
The upper, or application, layer uses the facilities of the lower layer as part of its operation. It relies on the lower layer to (i) locate a node responsible for a resource, (ii) notify it when nodes joined or left the system, so that it can manage the reallocation of resources to nodes. Many different kinds of applications can be implemented in the upper layer using the hashing provided by the lower layer. CFS and PAST both implement a read only file system on the routing layer. SQUIRREL [8] is a decentralized, peer-to-peer web caching system built on Pastry.

CFS uses Chord as its lower layer and implements a block level storage on it using a layer called Dhash. Each Chord node is assigned a 160-bit identifier number which is computed as the SHA-1 hash of its IP address and virtual server number (The virtual server number allows each Chord node to present multiple virtual servers and is used for load balancing). The 160-bit identifier space wraps around and is viewed as a circle. Chord identifiers are hence points on the circle (a simplified version with a 5-bit identifier space is shown in Figure 1). Blocks of data that are stored on Chord nodes are also assigned 160-bit identifiers that are computed has SHA-1 hashes of their content or some public key. Chord maps blocks to nodes responsible for them as follows: a block with Chord identifier number (or chordID) k is stored on the Chord node which has the smallest chordID greater than or equal to k. This node is denoted as the successor of k or $succ(k)$. The Chord layer on each node n maintains two pieces of information for routing:

1. The chordID and IP addresses of the first r nodes that follow n in the chord ring. The parameter r will be explained shortly.
2. A *finger table* with up to 160 entries, where the i^{th} entry holds the chordID and IP address of $succ(n + 2^{(i-1)})$. All arithmetic is modulo 2^{160}. A simplified version is shown in Figure 1 for a node in the 5-bit identifier space.

When a node n has to route a message to another node m, it first consults its list of successors to see if m is among them. If not, it uses its finger table to find the furthest finger, f, with chordID less than m. Node n then queries f about m. f either has m in its finger table, or refers n to one of its fingers that's closer to m.

The Dhash layer that is built on top of Chord manages storage. When a block, with chordID b, is to be stored in the network, Dhash uses Chord to find the node n ($n = succ(b)$) responsible for the block. It then contacts that node and instructs it to store the block. In order to maintain redundancy, copies of the block are also stored on the r successors of n (r is some suitable replication level). If n or any of its r successors depart, or if a new node joins and becomes a new successor of n, the Chord layer notifies the Dhash layer in these nodes. The Dhash layer then appropriately reassigns blocks. Block lookup is reasonably straightforward. When the Dhash layer in a node wants to find a block b, it uses the Chord layer to find $n = succ(b)$. It then contacts n or any of its r successors to retrieve the block (in case n is not available). One point to note is that when finding the successor of some chordID x, the Chord system first tries to find the first node preceding x on the ring (denoted as $pred(x)$). Once $pred(x)$ has been found, it can be queried to find its successor, which will be $succ(x)$.



Finger Table for 1

Finger #	Chord ID
1	5
2	5
3	5
4	14
5	19

Figure 1: Shows a Chord ring with a 5-bit identifier space. Nodes are denoted by circles and blocks are denoted by boxes. Both nodes and boxes are annotated with their ChordID. In addition blocks are annotated, in parenthesis, with the ID of their successor node. The finger table of node 1 is also shown. Row i of the node 1's finger table will hold the chordID and IP address of the node n, where $n = succ(1 + 2^{(i-1)})$, $i \in \{1, 2, 3, 4, 5\}$.

In PAST, Pastry provides the routing functionality. Like Chord, Pastry assigns nodes 128-bit identifiers (or IDs), that are treated as points in a circular address space. Pastry, however, uses a different scheme for mapping blocks to nodes. In Pastry a resource with ID x is assigned to r nodes, whose IDs are numerically closest to x (r is again a replication factor). Routing is also done differently. The routing table of a node n in has three sections (as shown in Figure 2:

1. A neighborhood set that holds the IDs and IP addresses of the nodes closest to n, by proximity, on this network.

2. A leaf set containing l entries. The entries comprise the IDs and IP addresses of $l/2$ nodes with the numerically closest IDs less than n and $l/2$ nodes with the numerically closest IDs greater than n. l is a system parameter.

3. A routing table with at most $(2^b - 1) \times \lceil \log_{2^b} N \rceil$ entries (N is the size of the identifier space). For the purpose of routing, IDs are thought of as a sequence of digits in the base 2^b. A node's routing table is organized into $\lceil \log_{2^b} N \rceil$ rows with $2^b - 1$ entries per row. The $2^b - 1$ entries at row i, refer to nodes whose ID matches n's ID up to the first i digits, but whose $(i+1)^{th}$ digit has one of the $2^b - 1$ possible values other than the $(i+1)^{th}$

NodeID: 10233102			
Leaf set:	Smaller	Larger	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232
Routing Table:			
−0−2212102	1	−2−2301203	−3−1203203
0	1−1−301233	1−2−230203	1−3−021022
10−0−31203	10−1−32102	2	10−3−23302
102−0−0230	102−1−1302	102−2−2302	3
1023−0−322	1023−1−000	1023−2−121	3
10233−0−01	1	10233−2−32	
0		102331−2−0	
		2	
Neighborhood set:			
13021022	10200230	11301233	31301233
02211202	22301203	31203203	33213321

Figure 2: State of a hypothetical Pastry node with ID 10233102. All numbers are in base 4 and each node keeps track of 8 leaves. The IDs of each entry in the routing table have been split as (common prefix with 10233102)-(next digit)-(rest of ID). As can be seen the top row has no common prefix with the node's ID and successive rows have larger prefixes in common. The IP addresses associated with each entry are not shown. Adapted from Rowstron et. al. [7].

digit of n. If there is more than one node available for an entry, then the closer node is chosen, based on some suitable proximity metric.

When node n receives a message that has to be routed to some node m, it checks to see if m is in its leaf set and routes the message directly, if so. If not, then then the message is forwarded to a node that shares a common prefix with m by at least one more digit (b bits) than it does. If no such node is available, it forwards the message to a node whose ID shares the same number of prefix digits as it does with m, but is numerically closer to m than it is. Hence the routing procedure always converges.

3 Weaknesses

A major issue in the peer-to-peer systems described above is that nodes may be unreliable. It is not possible to control their software or activities. Hence there is a very real possibility of certain parties deploying nodes that will try to make the system unavailable/unreliable. In this section, we consider weaknesses in such systems which can be exploited for disrupting operations like storing a block, lookup, ring formation etc.

A trivial way to break the system would be to have a large number of evil nodes join the network. They can coordinate among themselves to disrupt operations. For this project, however, we decided to assume that most nodes in the system were good and only a small fraction of them may be evil. This is consistent with the conditions under which reliability and availability are guaranteed in the CFS paper [5].

One important feature in both Chord and Pastry is the ability of a single physical node have more than one identity on the network. Evil nodes can take advantage of this in two ways. Firstly, if nodes can control where they appeared on the network (by picking one of the many available identities), they can choose these positions strategically e.g. a node can choose a position so that a particular data block gets mapped to it. Secondly, a node or group of nodes can simply instantiate all identities simultaneously and coordinate among these identities to disrupt operations.

In Chord, a node's identifier is computed as a cryptographic hash of its IP address and virtual server number Hence it can place itself at several positions depending on the number of IP addresses at its disposal and the number of virtual server numbers that the system allows a chord node to have.

Consider a Chord ring with $(n-1)$ nodes and one block. An evil node that wishes to control the block would have to make itself the successor of the block. If the evil node has at its disposal b possible identities ($b =$ number of IP's $\times$ number of virtual server nos.), the probability that it is able to become the block's successor using one of these b identities is:

$$p = 1 - \left(\frac{n-1}{n}\right)^b$$

Based on this we can compute the following:

$$\frac{b}{n} = \frac{\log(1-p)}{n[\log(n-1) - \log n]}$$

The fraction $\frac{b}{n}$ is almost constant for a given p. Hence if we wish to have certain odds at placing our evil node, then the number of evil identities we need is some constant fraction of the total number of nodes. Table 1 shows the number of identities needed for various probabilities in different sized networks. In Chord, the maximum number of virtual servers that can be started by a given node is an important factor that affects the ease with which an evil node can control its placement. In the prototype implementation from MIT, this value was set at 1024. Hence a node with a single IP available to it could place itself where it wanted in a network of size 1024 with probability of 0.5. The more the IP addresses available to a node, the more identities it can create and the larger the size of the network it can tackle.

A similar argument can made for the Pastry system. Given a network with $(n-1)$ nodes and a single block, if an evil node wants to control the block, the

Probability $\rightarrow$ Ring size $\downarrow$	$p = 0.25$	$p = 0.50$	$p = 0.75$
$n = 1024$	294	709	1418
$n = 16384$	9426	22712	45425
$n = 1048576$	301656	726817	1453634
$n = 1073741824$	308896273	744261117	1488522235

Table 1: Each entry in the table lists the number of identities an evil node must be able to create in order to have odds p of placing itself where it wants in a Chord network of size n

relations between p, b and n will be the same as before. However in Pastry, the number of identities (b) that an evil node can create is theoretically unlimited. The identifier for a node is computed as the cryptographic hash of its public key and the number of public keys a node can generate is unlimited.

4 Disrupting operations

In this section we describe the various attacks that we tried in Chord. These attacks assume that the evil node has been able to place itself suitably on the network. We tried two different attacks. The first attack was aimed at disrupting the store operation in Chord/Dhash and the second attack disrupted the lookup operation.

4.1 Disrupting stores

As mentioned earlier, in Chord, when a block is to be stored, Dhash first finds the node responsible for a block i.e. the blocks successor. It then performs a store operation on that node and its successors. We hoped that if an evil node was strategically placed as the successor of a block, the evil node could lie about storing the block and maintaining the replicas. The effectiveness of this attack however depends on exact implementation of the store. If the Dhash client requesting the store, sends the block to the successor and relies on the successor to perform replication, then successor can simply respond with an acknowledgment and not do anything. If the Dhash client first contacts the block's successor to find its r successors, it could instead send the block to each of them for storage. Even if we have a single evil node that would be in charge of the block, its successors would still hold replicas. The Chord prototype used the second approach. Hence, we were unable to disrupt the store operation. The PAST system, however, uses the first approach[6]. Hence it should be possible to disrupt the store operation in PAST.

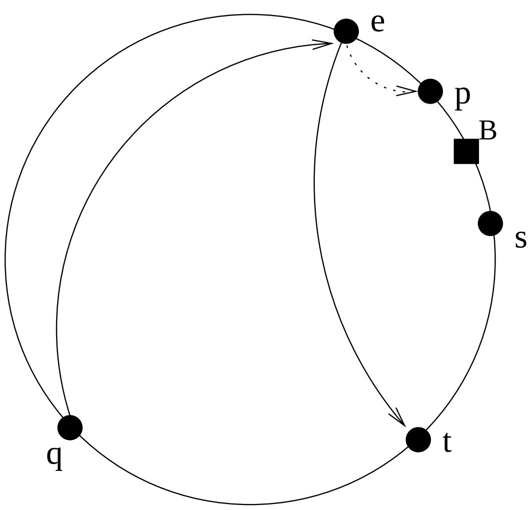


Figure 3: Disrupting the lookup in Chord

4.2 Disrupting the lookup

Lookup disruption is a weaker attack where evil nodes that are queried as part of a lookup lie about their fingers or successors. It is a weaker attack because it will only work if the evil node is queried as part of the lookup. In Pastry, the lookup is randomized to avoid making the path of a lookup deterministic. When a node receives a message destined for some other node m it finds an entry in its routing table that has a prefix of 1 digit more in common that it does with m. In reality multiple nodes are maintained at each entry. Hence during each lookup, a different node could be returned. The PAST implementation however, biases this selection toward nodes that are closer to the queried node.

In the Chord prototype, we were able to implement the lookup disruption successfully. Figure 3 shows the steps involved in a lookup operation. The points on the ring are nodes while the box is a block. Assume node q is searching for the block B. It makes a query to node e, to find out the closest finger of e that precedes B. Assume that e has p in its finger table (p is the predecessor of B). Ideally, e would return the id and IP of p to q (the dotted arc denotes this; e refers q to p during the lookup). q would then query p which would respond saying that it was the predecessor of b and return its successor's (i.e. t) ID and IP. We modified the Chord implementation to make e lie. When e was queried by q, e replied stating that it was the predecessor of B and it returned the ID and IP of t (the arc from e to t denotes this). t can be any node that lies beyond s and its r successors, but which lies before the finger f that follows e in $q's$ finger table. q would think that t was the successor of B and try fetching the block form t and its successors. t and its successors would simply respond stating that they don't have the block.

Ring size $\rightarrow$ No. of evil identities $\downarrow$	$n = 4096$	$n = 32768$	$n = 1048576$
$e = 32$	0.03227 (0.77%)	0.00507 (0.10%)	0.00021 (0.00%)
$e = 128$	0.12649 (3.03%)	0.02024 (0.39%)	0.00085 (0.01%)
$e = 1024$	0.85409 (20.0%)	0.15800 (3.03%)	0.00676 (0.10%)

Table 2: The table shows the odds of an evil node getting queried during a lookup of an n-node network. An evil node can create up to e identities. The number in the parenthesis is the percentage of nodes in the ring that are evil.

As mentioned earlier, this attack is weak, since the odds of an evil node being queried is quite low. In the Chord paper [4], they prove that with high probability, the number of nodes that get queried during a lookup in an n-node ring is $O(log(n))$. In the experimental section, they show that show that the average value is $\frac{1}{2}log(n)$. In a Chord system with n nodes and one evil node that can create up to e evil identities, the odds that a lookup would go through an evil node is:

$$p = e \times \frac{log(n+e)}{2(n+e)}$$

Table 2 shows the odds for different values of n and e. As can be seen, unless a significant fraction of the nodes in the ring are evil, the odds of the lookup going through an evil node are low.

5 Related work

There have been many papers that have addressed the issue of Byzantine failures in more traditional distributed systems [9, 10]. With the advent of peer-to-peer systems, researchers initially focused on improving the efficiency of these systems. Some recent projects have however looked at Byzantine failures in peer-to-peer systems too. Douceur [11] investigates, at an abstract level, the vulnerability of peer-to-peer systems to attacks where malicious hosts present several identities. It coined the name, *Sybil attack*, for such attacks. Liskov et. al. [12] present the design and implementation of a Chord like peer-to-peer system that is resilient against Sybil attacks. However the system they have proposed does place several restrictions on the base Chord design e.g.only trusted servers can join the ring. They use the peer-to-peer concept more as a way of avoiding centralization, unlike traditional peer-to-peer systems which are truly open. While there is a significant overlap of ideas in our work with Douceur[11] and Liskov [12], we have tried identifying actual attacks that could be mounted on second generation peer-to-peer systems.

6 Conclusion

In this project we had set out to investigate the vulnerability of peer-to-peer systems in the presence of Byzantine failures. We were able to identify a weakness that malicious hosts could exploit to attack the system, namely the ability of a host to present multiple identities. We have presented two possible attacks that relies on this weakness. We have also analyzed the feasibility of these attacks.

References

[1] `http://www.napster.com`

[2] `http://freenet.sourceforge.net`

[3] `http://www.gnutella.com`

[4] I. Stoica, R. Morris, D. Karger, M. F. Kashoek and H. Balakrishnan. Chord: A scalable, peer-to-peer lookup service for Internet applications. In Proc. ACM SIGCOMM'01 San Diego, CA. Aug. 2001.

[5] F. Dabek, M. F. Kashoek, D. Karger, R. Morris and I. Stoica. Wide-area cooperative storage with CFS. In Proc. ACM SOSP'01, Banff, Canada, Oct. 2001.

[6] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In Proc. HotOS VIII, Schloss Elmau, Germany, May 2001.

[7] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In Proc. IFIP/ACM Middleware 2001, Heidelberg, Germany. Nov. 2001

[8] S. Iyer, A. Rowstron and P. Druschel, SQUIRREL: A decentralized, peer-to-peer web cache, In Proc. PODC, Monterey, CA. 2002.

[9] L. Lamport, R. Shostak and M. Pease. The Byzantine Generals Problem. TPLS(43), 1982.

[10] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In Proc. OSDI, 1999.

[11] J. Douceur. The Sybil Attack. In Proc. IPTPS, Cambridge, Massachusetts. March 2002.

[12] R. Rodrigues, B. Liskov and L. Shrira. The Design of a Robust Peer-to-Peer System. In Proc. of the Tenth ACM SIGOPS European Workshop. Saint-Emilion, France, September 2002.