# A Survey of Peer-to-Peer Security Issues

Dan S. Wallach
dwallach@cs.rice.edu

Rice University, Houston, TX 77005, USA

**Abstract.** Peer-to-peer (p2p) networking technologies have gained popularity as a mechanism for users to share files without the need for centralized servers. A p2p network provides a scalable and fault-tolerant mechanism to locate nodes anywhere on a network without maintaining a large amount of routing state. This allows for a variety of applications beyond simple file sharing. Examples include multicast systems, anonymous communications systems, and web caches. We survey security issues that occur in the underlying p2p routing protocols, as well as fairness and trust issues that occur in file sharing and other p2p applications. We discuss how techniques, ranging from cryptography, to random network probing, to economic incentives, can be used to address these problems.

## 1 Introduction

Peer-to-peer systems, beginning with Napster, Gnutella, and several other related systems, became immensely popular in the past few years, primarily because they offered a way for people to get music without paying for it. However, under the hood, these systems represent a paradigm shift from the usual web client/server model, where there are no "servers;" every system acts as a peer, and by virtue of the huge number of peers, objects can be widely replicated, providing the opportunity for high availability and scalability, despite the lack of centralized infrastructure.

Capitalizing on this trend, researchers have defined structured peer-to-peer (p2p) overlays such as CAN [1], Chord [2], Pastry [3] and Tapestry [4] provide a self-organizing substrate for large-scale p2p applications. Unlike earlier systems, these have been subject to more extensive analysis and more careful design to guarantee scalability and efficiency. Also, rather than being designed specifically for the purpose of sharing unlawful music, these systems provide a powerful platform for the construction of a variety of decentralized services, including network storage, content distribution, web caching, searching and indexing, and application-level multicast. Structured overlays allow applications to locate any object in a probabilistically bounded, small number of network hops, while requiring per-node routing tables with only a small number of entries. Moreover, the systems are scalable, fault-tolerant and provide effective load balancing.

Making these systems "secure" is a significant challenge [5, 6]. In general, any system not designed to withstand an adversary is going to be broken easily by one, and p2p systems are no exception. If p2p systems are to be widely deployed on the Internet (at least, for applications beyond sharing "pirate" music files), they must be robust against a conspiracy of some nodes, acting in concert, to attack the remainder of the nodes. A malicious node might give erroneous responses to a request, both at the application

level (returning false data to a query, perhaps in an attempt to censor the data) or at the network level (returning false routes, perhaps in an attempt to partition the network). Attackers might have a number of other goals, including traffic analysis against systems that try to provide anonymous communication, and censorship against systems that try to provide high availability.

In addition to such "hard" attacks, some users may simply wish to gain more from the network than they give back to it. Such disparities could be expressed in terms of disk space (where an attacker wants to store more data on p2p nodes than is allowed on the attacker's home node), or in terms of bandwidth (where an attacker refuses to use its limited network bandwidth to transmit a file, forcing the requester to use some other replica). While many p2p applications are explicitly designed to spread load across nodes, "hot-spots" can still occur, particularly if one node is responsible for a particularly popular document.

Furthermore, a number of "trust" issues occur in p2p networks. As new p2p applications are designed, the code for them must be deployed. In current p2p systems, the code to implement the p2p system must be trusted to operate correctly; p2p servers typically execute with full privileges to access the network and hard disk. If arbitrary users are to create code to run on p2p systems, an architecture to safely execute untrusted code must be deployed. Likewise, the data being shared, itself, might not be trustworthy. Popularity-based ranking systems will be necessary to help users discover documents that they desire.

Of course, many other issues exist that could be classified as security issues that will not be considered in this paper. For example, one pressing problem with the Kazaa system, often used to share pirated music and movies, is its use of bandwidth [7], which has led many ISPs and universities to either throttle the bandwidth or ban these systems outright. Likewise, this paper only considers security for one high-level p2p application: sharing files. There are numerous other possible applications that can be built using p2p systems (e.g., event notification systems [8, 9]), which would have their own security issues.

The remainder of this paper is a survey of research in these areas. Section 3 discusses correctness issues in p2p routing. Section 4 discusses correctness and fairness issues in p2p data storage and file sharing. Section 5 discusses trust issues. Section 6 presents related work and Section 7 has conclusions.

## 2 Background, models and solution

In this section, we present some background on structured p2p overlay protocols like CAN, Chord, Tapestry and Pastry. Space limitations prevent us from giving a detailed overview of each protocol. Instead, we describe an abstract model of structured p2p overlay networks that we use to keep the discussion independent of any particular protocol. For concreteness, we also give an overview of Pastry and point out relevant differences between it and the other protocols. Next, we describe models and assumptions used later in the paper about how nodes might misbehave. Finally, we define secure routing and outline our solution.

Throughout this paper, most of the analyses and techniques are presented in terms of this model and should apply to other structured overlays except when otherwise noted. However, the security and performance of our techniques was fully evaluated only in the context of Pastry; a full evaluation of the techniques in other protocols is future work.

### 2.1 Routing overlay model

We define an abstract model of a structured p2p routing overlay, designed to capture the key concepts common to overlays such as CAN, Chord, Tapestry and Pastry.

In our model, participating nodes are assigned uniform random identifiers, *nodeIds*, from a large *id space* (e.g., the set of 128-bit unsigned integers). Application-specific objects are assigned unique identifiers, called *keys*, selected from the same id space. Each key is mapped by the overlay to a unique live node, called the key's *root*. The protocol routes messages with a given key to its associated root.

To route messages efficiently, all nodes maintain a *routing table* with the nodeIds of several other nodes and their associated IP addresses. Moreover, each node maintains a *neighbor set*, consisting of some number of nodes with nodeIds nearest itself in the id space. Since nodeId assignment is random, any neighbor set represents a random sample of all participating nodes.

For fault tolerance, application objects are stored at more than one node in the overlay. A *replica function* maps an object's key to a set of *replica keys*, such that the set of *replica roots* associated with the replica keys represents a random sample of participating nodes in the overlay. Each replica root stores a copy of the object.

Next, we discuss existing structured p2p overlay protocols and how they relate to our abstract model.

### 2.2 Pastry

Pastry nodeIds are assigned randomly with uniform distribution from a circular 128-bit id space. Given a 128-bit key, Pastry routes an associated message toward the live node whose nodeId is numerically closest to the key. Each Pastry node keeps track of its neighbor set and notifies applications of changes in the set.

**Node state:** For the purpose of routing, nodeIds and keys are thought of as a sequence of digits in base $2^b$ ($b$ is a configuration parameter with typical value 4). A node's routing table is organized into $128/2^b$ rows and $2^b$ columns. The $2^b$ entries in row $r$ of the routing table contain the IP addresses of nodes whose nodeIds share the first $r$ digits with the given node's nodeId; the $r+1$th nodeId digit of the node in column $c$ of row $r$ equals $c$. The column in row $r$ that corresponds to the value of the $r+1$th digit of the local node's nodeId remains empty. A routing table entry is left empty if no node with the appropriate nodeId prefix is known. Figure 1 depicts an example routing table.

Each node also maintains a neighbor set. The neighbor set is the set of $l$ nodes with nodeIds that are numerically closest to a given node's nodeId, with $l/2$ larger and $l/2$ smaller nodeIds than the given node's id. The value of $l$ is constant for all nodes in the overlay, with a typical value of approximately $\lceil 8 * log_{2^b} N \rceil$, where $N$ is the number of

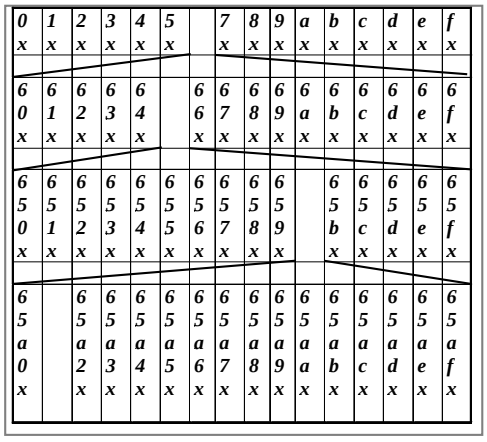

**Fig. 1.** Routing table of a Pastry node with nodeId 65$a$1$x$, $b = 4$. Digits are in base 16, $x$ represents an arbitrary suffix.

expected nodes in the overlay. The leaf set ensures reliable message delivery and is used to store replicas of application objects.

**Message routing:** At each routing step, a node seeks to forward the message to a node in the routing table whose nodeId shares with the key a prefix that is at least one digit (or $b$ bits) longer than the prefix that the key shares with the current node's id. If no such node can be found, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node, but is numerically closer to the key than the current node's id. If no appropriate node exists in either the routing table or neighbor set, then the current node or its immediate neighbor is the message's final destination.

Figure 2 shows the path of an example message. Analysis shows that the expected number of routing hops is slightly below $log_{2^b}N$, with a distribution that is tight around the mean. Moreover, simulation shows that the routing is highly resilient to crash failures.

To achieve self-organization, Pastry nodes must dynamically maintain their node state, i.e., the routing table and neighbor set, in the presence of node arrivals and node failures. A newly arriving node with the new nodeId $X$ can initialize its state by asking any existing Pastry node $A$ to route a special message using $X$ as the key. The message is routed to the existing node $Z$ with nodeId numerically closest to $X$. $X$ then obtains the neighbor set from $Z$ and constructs its routing table by copying rows from the routing tables of the nodes it encountered on the original route from $A$ to $Z$. Finally, $X$ announces its presence to the initial members of its neighbor set, which in turn update their own neighbor sets and routing tables. Similarly, the overlay can adapt to abrupt node failure by exchanging a small number of messages ($O(log_{2^b}N)$) among a small number of nodes.

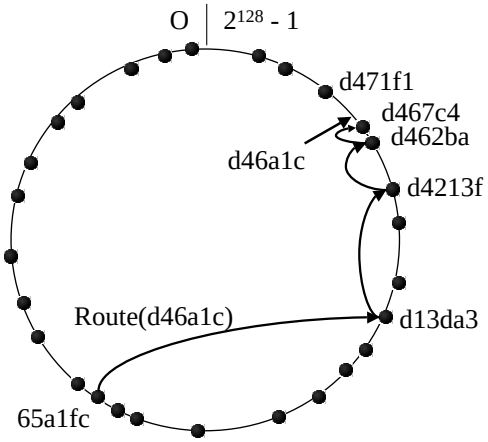


**Fig. 2.** Routing a message from node 65*a*1*fc* with key *d*46*a*1*c*. The dots depict live nodes in Pastry's circular namespace.

### 2.3 CAN, Chord, Tapestry

Next, we briefly describe CAN, Chord and Tapestry, with an emphasis on the differences relative to Pastry.

**Tapestry** is very similar to Pastry but differs in its approach to mapping keys to nodes and in how it manages replication. In Tapestry, neighboring nodes in the namespace are not aware of each other. When a node's routing table does not have an entry for a node that matches a key's *n*th digit, the message is forwarded to the node with the next higher value in the *n*th digit, modulo $2^b$, found in the routing table. This procedure, called *surrogate routing*, maps keys to a unique live node if the node routing tables are consistent. Tapestry does not have a direct analog to a neighbor set, although one can think of the lowest populated level of the Tapestry routing table as a neighbor set. For fault tolerance, Tapestry's replica function produces a set of random keys, yielding a set of replica roots at random points in the id space. The expected number of routing hops in Tapestry is $log_{2^b}N$.

**Chord** uses a 160-bit circular id space. Unlike Pastry, Chord forwards messages only in clockwise direction in the circular id space. Instead of the prefix-based routing table in Pastry, Chord nodes maintain a routing table consisting of up to 160 pointers to other live nodes (called a "finger table"). The *i*th entry in the finger table of node *n* refers to the live node with the smallest nodeId clockwise from $n+2^{i-1}$. The first entry points to *n*'s successor, and subsequent entries refer to nodes at repeatedly doubling distances from *n*. Each node in Chord also maintains pointers to its predecessor and to its *n* successors in the nodeId space (this successor list represents the neighbor set in our model). Like Pastry, Chord's replica function maps an object's key to the nodeIds in the neighbor set of the key's root, i.e., replicas are stored in the neighbor set of the key's root for fault tolerance. The expected number of routing hops in Chord is $\frac{1}{2}log_2N$.

**CAN** routes messages in a *d*-dimensional space, where each node maintains a routing table with $O(d)$ entries and any node can be reached in $(d/4)(N^{1/d})$ routing hops on

average. The entries in a node's routing table refer to its neighbors in the $d$-dimensional space. CAN's neighbor table duals as both the routing table and the neighbor set in our model. Like Tapestry, CAN's replica function produces random keys for storing replicas at diverse locations. Unlike Pastry, Tapestry and Chord, CAN's routing table does not grow with the network size, but the number of routing hops grows faster than $logN$ in this case.

Tapestry and Pastry construct their overlay in a Internet topology-aware manner to reduce routing delays and network utilization. In these protocols, routing table entries can be chosen arbitrarily from an entire segment of the nodeId space without increasing the expected number of routing hops. The protocols exploit this by initializing the routing table to refer to nodes that are nearby in the network topology and have the appropriate nodeId prefix. This greatly facilitates proximity routing [3]. However, it also makes these systems vulnerable to attacks where an attacker can exploit his locality to the victim.

The choice of entries in CAN's and Chord's routing tables is tightly constrained. The CAN routing table entries refer to specific neighboring nodes in each dimension, while the Chord finger table entries refer to specific points in the nodeId space. This makes proximity routing harder, but it protects nodes from attacks that exploit attacking nodes' proximity to their victims.

### 2.4 System model

The system runs on a set of $N$ nodes that form an overlay using one of the protocols described in the previous section. We assume a bound $f$ ($0 \leq f < 1$) on the fraction of nodes that may be faulty. Faults are modeled using a constrained-collusion Byzantine failure model, i.e., faulty nodes can behave arbitrarily and they may not all necessarily be operating as a single conspiracy. The set of faulty nodes is partitioned into independent coalitions, which are disjoint sets with size bounded by $cN$ ($1/N \leq c \leq f$). When $c = f$, all faulty nodes may collude with each other to cause the most damage to the system. We model the case where faulty nodes are grouped into multiple independent coalitions by setting $c < f$. Members of a coalition can work together to corrupt the overlay but are unaware of nodes in other coalitions. We studied the behavior of the system with $c$ ranging from $1/N$ to $f$ to model different failure scenarios.

We assume that every node in the p2p overlay has a static IP address at which it can be contacted. In this paper, we ignore nodes with dynamically assigned IP addresses, as well as nodes behind network address translation boxes or firewalls. While p2p overlays can be extended to address these concerns, this paper focuses on more traditional network hosts.

All nodes communicate over normal Internet connections. We distinguish between two types of communication: *network-level*, where nodes communicate directly without routing through the overlay, and *overlay-level*, where messages are routed through the overlay using one of the protocols discussed in the previous section. We use cryptographic techniques to prevent adversaries from observing or modifying network-level communication between legitimate nodes. An adversary has complete control over network-level communication to and from nodes that it controls. This gives an adversary an opportunity to observe and either discard or misroute traffic through faulty nodes it

controls. If the messages are protected by appropriate cryptography, then modifications to them should be detected. Some messages, such as routing updates, may not be easily amenable to the application of cryptographic techniques.

## 3 Routing in p2p systems

The routing primitives implemented by current structured p2p overlays provide a best-effort service to deliver a message to a replica root associated with a given key. As discussed above, a malicious overlay node has ample opportunities to corrupt overlay-level communication. Therefore, these primitives are not sufficient to construct secure applications. For example, when inserting an object, an application cannot ensure that the replicas are placed on legitimate, diverse replica roots as opposed to faulty nodes that impersonate replica roots. Even if applications use cryptographic methods to authenticate objects, malicious nodes may still corrupt, delete, deny access to or supply stale copies of all replicas of an object.

To address this problem, we must create a secure routing primitive. *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.* $R_k$ is defined as the set of nodes that contains, for each member of the set of replica keys associated with $k$, a live root node that is responsible for that replica key. In Pastry, for instance, $R_k$ is simply a set of live nodes with nodeIds numerically closest to the key. Secure routing ensures that (1) the message is eventually delivered, despite nodes that may corrupt, drop or misroute the message; and (2) the message is delivered to all legitimate replica roots for the key, despite nodes that may attempt to impersonate a replica root.

Secure routing can be combined with existing security techniques to safely maintain state in a structured p2p overlay. For instance, *self-certifying data* can be stored on the replica roots, or a Byzantine-fault-tolerant replication algorithm [10] can be used to maintain the replicated state. Secure routing guarantees that the replicas are initially placed on legitimate replica roots, and that a lookup message reaches a replica if one exists. Similarly, secure routing can be used to build other secure services, such as maintaining file metadata and user quotas in a distributed storage utility. The details of such services are beyond the scope of this paper.

Implementing the secure routing primitive requires the solution of three problems: securely assigning nodeIds to nodes, securely maintaining the routing tables, and securely forwarding messages. Secure nodeId assignment ensures that an attacker cannot choose the value of nodeIds assigned to the nodes that the attacker controls. Without it, the attacker could arrange to control all replicas of a given object, or to mediate all traffic to and from a victim node.

Secure routing table maintenance ensures that the fraction of faulty nodes that appear in the routing tables of correct nodes does not exceed, on average, the fraction of faulty nodes in the entire overlay. Without it, an attacker could prevent correct message delivery, given only a relatively small number of faulty nodes. Finally, secure message forwarding ensures that at least one copy of a message sent to a key reaches each correct

replica root for the key with high probability. These techniques are described in greater detail in Castro et al. [5], but are outlined here.

### 3.1 Secure nodeId assignment

In the original design of Pastry, and in many other p2p systems, nodeIds are chosen at random from the space of all identifiers (i.e., for Pastry, a randomly chosen 128-bit number). The problem with such a system is that a node might choose its identifier maliciously. A coalition of malicious nodes that wishes to censor a specific document could easily allocate itself a collection of nodeIds closer to that document's key than any existing nodes in the system. This would allow the coalition to control all the replica roots for that document, giving them the ability to censor the document from the network. Likewise, a coalition could similarly choose nodeIds to maximize its chances of appearing in a victim node's routing tables. If all the outgoing routes from a victim point to nodes controlled by the coalition, then *all* of the victim's access to the overlay network is mediated (and possibly censored) by the coalition. It's necessary, therefore, to guarantee that nodeIds are assigned randomly.

The simplest design to perform secure nodeId assignments is to have a centralized authority that produces cryptographic nodeId certificates, a straightforward extension to standard cryptographic techniques: rather than binding an e-mail address to a public key, these certificates instead bind a nodeId, chosen randomly by the server, to a public key generated by the client machine. The server is only consulted when new nodes join and is otherwise uninvolved in the actions of the p2p system. As such, such a server would have no impact on the scalability or reliability of the p2p overlay.

Regardless, to make such a design work, we must concern ourselves with Sybil attacks [11], wherein a hostile node or coalition of nodes might try to get a large number of nodeIds. Even if those nodeIds are random, a large enough collection of them would still give the attackers disproportionate control over the network. The best solution we currently have to this problem is to moderate the *rate* at which nodeIds are given out. Possible solutions include charging money in return for certificates or requiring some form of external authentication. While it may be possible to use some form of cryptographic puzzles [12], these still allow attackers with large computational resources to get a disproportionate number of nodeIds.

An open problem is assigning random nodeIds without needing a centralized authority. We considered a number of possibilities, including variations on cryptographic puzzles and multi-party bit-commitment schemes. Unfortunately, all such schemes appear to open the possibility that an attacker can rejoin the network, repeatedly, and eventually gain an advantage.

### 3.2 Robust routing primitives

Even with perfect nodeId assignment, when an attacker controls a fraction $f$ of the nodes in the p2p network, we would expect that each entry in every routing table would have a probability of $f$ of pointing to a malicious node. If a desired route consumes $h$ hops, then the odds of a complete route being free of malicious nodes is $(1-f)^h$. In practice, with Pastry, if 50% of nodes are malicious, then the probability of a route

reaching the correct destination ranges between about 50% for overlay networks with 1000 nodes, to about 25% for overlay networks with 1000000 nodes. These odds assume, however, that an adversary cannot increase its probability of being on a given route. However, if the adversary could take advantage of its locality to a given victim node to get more entries in that node's routing table, then the adversary could increase its odds of controlling any given route that uses the victim node.

To prevent locality-based attacks, we introduced a technique called *constrained routing*, which trades off locality vs. performance. Where Pastry normally tries to fill the routing table with "local" nodes (i.e., low latency, high bandwidth) having the necessary nodeIds, a constrained routing table insists on having the closest nodes, in nodeId space, to keys which have the necessary prefix and the same suffix as the node itself.

Next, we would like to increase the odds of a message reaching the desired replica roots beyond the $(1-f)^h$, described above. To do this, we can attempt multiple, redundant routes from the source to the destination. In Pastry, we do this by sending the message from the source node to all of its neighbors in the p2p overlay. Because nodeIds are random, the neighbors should represent a random, geographically diverse, sampling of the nodes in the p2p overlay. From there, each neighbor node forwards the message toward the target node. If at least one of the neighbors can achieve a successful route, then the message is considered successfully delivered. Based on modeling and corroborated with simulations, we have measured that this operation can be successful with a 99.9% probability, as long as $f \leq 30\%$.

### 3.3 Ejecting misbehaving nodes

Our existing models and simulations show Pastry can route successfully when as many as 30% of the nodes in the p2p overlay network are malicious. However, it would be preferable to have mechanisms to actively *remove* malicious nodes when they are detected. An interesting open problem is how to remove a malicious node from the overlay. While all p2p overlays must have provisions for recovering when a node fails, we would like these mechanisms to be invocable when a node is still alive and functioning. When one node accuses another of cheating, there needs to be some way that it can *prove* its accusation, in order to convince other nodes to eject the malicious node from the network.

While such a proof may be generated at the application layer (see the discussion in Section 4.2), it's not clear how such a proof could be generated at the routing layer. If a node is simply dropping messages with some probability or is pretending that perfectly valid nodes do not exist, such behavior could also be explained by failures in the underlying Internet fabric. Addressing this, in general, is an interesting open problem.

## 4 Storage

In the following, we describe how applications can securely maintain state while minimizing the use of secure routing for performance reasons. A common approach to reduce reliance on secure routing, when reading an object, is to store *self-certifying data* in the overlay. For example, CFS [13] uses a cryptographic hash of a file's contents as

the key during insertion and lookup of the file, and PAST [14] inserts signed content into the overlay. This allows the client to use insecure, more efficient routing to retrieve a copy of a file for reading. When the client receives a copy of the file, it checks its integrity. If the client fails to receive a copy of the file or if the integrity check fails, then the client can use secure routing to retrieve a (hopefully) correct copy or to verify that the file is simply unavailable in the overlay. Of course, it is important to use secure routing for any object insertions because, otherwise, all replicas of the new version may be stored on faulty nodes.

Self-certifying data is a useful solution only when the client knows a hash for the document it's looking for. Even with self-certifying path names [15], or other forms of Merkle hash trees [12] where the user has a secure hash of the document being requested before it is loaded, the user must trust the origin of that secure hash. If the user is using some kind of search engine, cryptographic techniques cannot prevent undesirable documents appearing in the list of search results. And, no amount of cryptographic integrity checking can prevent denial of service attacks.

However, once these problems are considered, the issue of *incentives* emerges as the predominant problem for multiple users sharing their disk space with one another. Why should one computer user allow her disk space and network bandwidth to be used by another user, somewhere else? If possible, she might prefer to contribute nothing for the common good, and consume others' resources without paying them. To prevent such a *tragedy of the commons*, the system must be designed to limit how much remote space one can consume without providing a suitable amount of storage for the use of others.

### 4.1 Quota Architectures

In the Farsite study [16], the authors noted that hard drives are often relatively empty. As hard drives grow larger and larger, this trend seems likely to continue. If the goal is to create a distributed storage system using this empty space, an interesting fairness issue occurs. A malicious node might choose to claim its storage is full, when it actually has free space. Or, more generally, it might wish to use more storage from remote nodes than it provides for the use of others in the p2p system. Our goal is to create a quota system that guarantees equitable sharing of these resources.

As with nodeId assignment, a simple solution is to require the use of a universally trusted quota authority. However, in the nodeId assignment problem, the nodeId authority need only be consulted when a new node wishes to acquire a nodeId. Otherwise, the nodeId authority need not be involved in the activity of the p2p network. With a centralized quota authority, every request to store a document would require a query to the quota authority. This would create a huge bottleneck as the size of the p2p overlay scaled up.

To distribute this authority, the original design of PAST [14] hypothesized that a smart card would be attached to each node of the network. The smartcards would be responsible for tracking each node's use of remote resources and would issue digitally signed tickets, allowing the local node to prove to a remote node that it was under its quota. Of course, it may not be practical to issue smartcards to millions of nodes. Furthermore, if p2p users can compromise the key material inside their smartcards, they would gain effectively unlimited storage within the p2p overlay.

An alternative architecture would be to ask a node's neighbors to act as *quota managers* on behalf of the node. Collectively, a node's neighbors can act together to endorse a node's request to store a document in the same way as the local smartcard might. The quota information would be distributed and replicated among the neighbors in precisely the same was as any other data in PAST. The main weakness with this scheme is that the quota managers do not have any particular *incentive* to participate in maintaining the quota information. It would be cheaper to track nothing and always endorse a request. Ideally, we would like to create a system where nodes have a natural economic incentive to keep track of each other's disk storage. This is an instance of a problem in distributed algorithm mechanism design [17].

### 4.2 Distributed Auditing

We can look at disk space as a commodity, and the sharing of disk space in a p2p overlay network as a barter economy of disk space. Nodes trade the use of their local storage for the use of other nodes' remote storage. What mechanisms can be used to implement such an economy? We are currently studying the use of *auditing*. In our system, each node publishes, and digitally signs, two logs: the *local list* of files that the local node is storing on behalf of remote nodes, and the *remote list* of files that other nodes are storing on behalf of the local node. Each entry in the logs contains the name of the remote node responsible and the size of the object being stored. Also, the local list contains the amount of free space available on the local node.

This now creates a nicely balanced system. If a node, *A* wishes to store a file on *B*, then *B* need only read *A*'s logs to make sure that *A* is using less resources than it is providing.

In general, when *B* is storing a file on behalf of *A*, *B* has an incentive to audit *A* to make sure that *A* is "paying" for its storage. If *A* does not list the file in its remote list, then it's not "paying" anything to *B*; *B* should therefore feel free to delete the file. Likewise, *A* has an incentive to audit *B*, to make sure that *B* is actually storing the file, versus quietly dropping the file and perhaps relying on the other replicas to maintain the file. If *A* queries *B* for random portions of its file (while first alerting any other replicas that an audit is under way) and *B* cannot answer, then *A* can remove *B* from its remote list; there's no reason for *A* to pay for service it's not using.

But, what if *A* wishes to lie to *B*, feeding it a log that understates its remote storage usage? To address this, we need anonymous communication. Luckily, many architectures are available to do this. In particular, Crowds [18] maps very easily onto a p2p overlay network. So long as audits are timed randomly, where *A* does not know whether the node checking on it is *B* or perhaps some other node with which it's done business, then *A* cannot customize its logs to present itself in a better light to *B*. Or, if it did, the signed log forms a digital "confession" of its misbehavior when compared with the logs it sends to other nodes.

Of particular interest, once we've created this disk economy, is that we now have mechanisms suitable for applying peer pressure. Fehr and Gachter [19] have shown that people are willing to spend money to eject cheaters from an economy, allowing the economy to quickly reach a stable state, free of cheaters. This auditing system allows for nodes to "spend" disk space simply by increasing the size of the remote list, thereby

"paying" for somebody to be ejected. Combining that with any "confessions" from misbehaving nodes, and the system appears to provide strong disincentives to cheaters.

One remaining issue is what we call "cheating chains." It's possible for one node to push its deficits off its own books by conspiring with other nodes. *A* can claim it is storing a large file on behalf of *B*. So long as *B* claims it's storing the file on *A*, the audit logs check out. If *A* and *B* are conspiring together, then no actual files need be stored. Furthermore, imagine that *B* claims its storing a file on behalf of *C*, and *C* claims it's storing a file on behalf of *D*. Again, when they're all conspiring, nobody need actually store any files, and the only way somebody might detect that *A* were cheating would be to audit *A*, then *B*, then *C*, and finally *D* before detecting, perhaps, that *D*'s books were out of balance. The best solution we have to this problem is for all nodes in the p2p overlay to perform *random audits*, choosing a key, at random, and auditing the node with the closest nodeId, comparing that node's logs to the logs of every node with which it claims it's sharing. If every node chooses another node at random, on a regular basis, then every node will be audited with a very high probability. Our current simulations show that the cost of this auditing is quite reasonable, consuming an aggregate bandwidth of only 100-200 bits/second, even in large p2p overlays, although this bandwidth does increase with the size of the logs.

### 4.3 Other Forms of Fairness

This paper has focused primarily on fair sharing of disk space, but there are many other aspects to fair sharing. In particular, we would like to guarantee fair sharing of network bandwidth. In current p2p networks, nodes can easily find themselves hosting a huge amount of traffic on behalf of other nodes, even while making very little use of the network on their own behalf. With the Kazaa system, in particular, the bandwidth generated by some nodes has been enough to force many universities to use traffic shaping technologies to prevent student machines, running Kazaa, from overwhelming the campus's limited bandwidth to the Internet.

One conceivable solution would require the use of micropayment systems. When a user wishes to query the p2p overlay, that would require spending a token. When the user's machine receives a query, it also receives a token that it can use later. If a given machine has more tokens than it needs, perhaps it would refuse to service any queries. Unfortunately, it's not clear whether any current micropayment schemes scale to support so many nodes making so many small queries of each other. While it might be possible to add these bandwidth tokens onto the audit logs described above, the cost of evaluating whether a token is valid could be significantly greater than simply servicing the request without checking the token's validity.

Another issue, assuming the token scheme can be made to work, would be suitably redesigning file sharing to preserve the availability of data. In effect, we would allow nodes to deliberately fail to service requests because they had no more need for tokens. To compensate for this, data will need to be much more widely replicated than in traditional p2p overlays.

## 5 Trust in p2p overlays

P2p systems generally require a remarkable amount of trust from their participants. A node must trust that other nodes implement the same protocols and will respect the goals of the system. In previous sections, we have discussed how mechanisms can be developed to work around a certain percent of the nodes violating the rules, but there are many other aspects where trust issues arise.

*Popularity* When documents are requested based on keywords, rather than cryptographically strong hashes, it becomes possible for an adversary to spoof the results. The recording industry, in particular, has apparently been deploying "decoy" music files in p2p networks that have the same name as music files by popular artists. The decoy files have approximately the correct length, but do not contain the desired music. Similar issues have traditionally hurt search engines, where any page with a given search term inside it had an equal chance of appearing highly on the search results. The best solution to the search engine problem, as used by Google's PageRank technology, has been to form a notion of popularity. For Google, pages that are linked from "popular" pages are themselves more popular. An interesting issue is how to add such a notion of popularity into a p2p storage system. It might be possible to extend the audit logs, from Section 4.2, to allow nodes to indicate the value, or lack thereof, of a given file. If users can then rank each others rankings, this could potentially allow the creation of a system comparable to Google's PageRank.

*Code* Fundamentally, p2p systems require the user to install a program on their computer that will work with other p2p nodes to implement the system. Since many applications can be built on a generic p2p substrate, an interesting issue becomes how to distribute the code to support these p2p applications. Users should not necessarily trust arbitrary programs, written by third parties, to run on their system. Recently, some commercial p2p systems were discovered to redirect sales commissions from online purchases to the p2p developers [20] and might also sell the use of CPU cycles on a user's computer to third parties, without the user getting any reimbursement [21]. Why should a user arbitrarily grant such privileges to p2p code? In many respects, this same problem occurred with active networks [22], except, in those systems, the computational model could be restricted [23]. For p2p systems, where applications can perform significant computations and consume vast amounts of disk storage, it would appear that a general-purpose mobile code security architecture [24] is necessary.

## 6 Related work

P2p systems have been designed in the past to address numerous security concerns, providing anonymous communication, censorship resistance, and other features. Many such systems, including onion routing [25], Crowds [18], Publius [26], and Tangler [27], fundamentally assume a relatively small number of nodes in the network, all well-known to each other. To scale to larger numbers of nodes, where it is not possible to maintain a canonical list of the nodes in the network, additional mechanisms are

necessary. Some recent p2p systems have also been developed to support censorship resistance [28] and anonymity [29, 30].

Sit and Morris [6] present a framework for performing security analyses of p2p networks. Their adversarial model allows for nodes to generate packets with arbitrary contents, but assumes that nodes cannot intercept arbitrary traffic. They then present a taxonomy of possible attacks. At the routing layer, they identify node lookup, routing table maintenance, and network partitioning / virtualization as security risks. They also discuss issues in higher-level protocols, such as file storage, where nodes may not necessarily maintain the necessary invariants, such as storage replication. Finally, they discuss various classes of denial-of-service attacks, including rapidly joining and leaving the network, or arranging for other nodes to send bulk volumes of data to overload a victim's network connection (i.e., distributed denial of service attacks).

Dingledine *et al.* [31] and Douceur [11] discuss address spoofing attacks. With a large number of potentially malicious nodes in the system and without a trusted central authority to certify node identities, it becomes very difficult to know whether you can trust the claimed identity of somebody with whom you have never before communicated. Dingledine proposes to address this with various schemes, including the use of micro-cash, that allow nodes to build up *reputations*.

Bellovin [32] identifies a number of issues with Napster and Gnutella. He discusses how difficult it might be to limit Napster and Gnutella use via firewalls, and how they can leak information that users might consider private, such as the search queries they issue to the network. Bellovin also expresses concern over Gnutella's "push" feature, intended to work around firewalls, which might be useful for distributed denial of service attacks. He considers Napster's centralized architecture to be more secure against such attacks, although it requires all users to trust the central server.

## 7 Conclusions

This paper has surveyed some security issues that occur in peer-to-peer overlay networks, both at the network layer and at the application layer. We have shown how techniques ranging from cryptography through redundant routing to economic methods can be applied to increase the security, fairness, and trust for applications on the p2p network. Because of the diversity of how p2p systems are used, there will be a corresponding diversity of security solutions applied to the problems.

## Acknowledgments

This paper draws on conversations, speculations, and joint research with many of my colleagues. I specifically thank Peter Druschel, Antony Rowstron, Ayalvadi Ganesh, and Miguel Castro, with whom I have studied techniques for securing p2p overlay routing, and Tsuen Wan "Johnny" Ngan, with whom I've been working on auditing architectures for p2p file storage. I also thank Tracy Volz for suggestions on this work. This work was supported in part by NSF grant CCR-9985332.

## References


1. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: Proc. ACM SIGCOMM'01, San Diego, California (2001)
2. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for Internet applications. In: Proc. ACM SIGCOMM'01, San Diego, California (2001)
3. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proc. IFIP/ACM Middleware 2001, Heidelberg, Germany (2001)
4. Zhao, B.Y., Kubiatowicz, J.D., Joseph, A.D.: Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley (2001)
5. Castro, M., Druschel, P., Ganesh, A., Rowstron, A., Wallach, D.S.: Secure routing for structured peer-to-peer overlay networks. In: Proc. OSDI 2002, Boston, Massachusetts (2002) To appear.
6. Sit, E., Morris, R.: Security considerations for peer-to-peer distributed hash tables. In: Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, Massachusetts (2002)
7. Saroiu, S., Gummadi, K.P., Dunn, R.J., Gribble, S.D., Levy, H.M.: An analysis of internet content delivery systems. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, Massachusetts (2002)
8. Rowstron, A., Kermarrec, A.M., Druschel, P., Castro, M.: Scribe: The design of a large-scale event notification infrastructure. In: Proc. NGC'2001, London, UK (2001)
9. Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: SCRIBE: A large-scale and decentralized application-level mul ticast infrastructure. IEEE JSAC **20** (2002)
10. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99), New Orleans, Louisiana (1999)
11. Douceur, J.R.: The Sybil attack. In: Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, Massachusetts (2002)
12. Merkle, R.C.: Secure communications over insecure channels. Communications of the ACM **21** (1978) 294–299
13. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. In: Proc. ACM SOSP'01, Banff, Canada (2001)
14. Rowstron, A., Druschel, P.: Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In: Proc. ACM SOSP'01, Banff, Canada (2001)
15. Mazières, D., Kaminsky, M., Kaashoek, M.F., Witchel, E.: Separating key management from file system security. In: Proc. SOSP'99, Kiawah Island, South Carolina (1999)
16. Bolosky, W.J., Douceur, J.R., Ely, D., Theimer, M.: Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In: Proc. SIGMETRICS'2000, Santa Clara, California (2000)
17. Feigenbaum, J., Shenker, S.: Distributed algorithmic mechanism design: Recent results and future directions. In: Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M 2002), Atlanta, Georgia (2002) 1–13
18. Reiter, M.K., Rubin, A.D.: Anonymous Web transactions with Crowds. Communications of the ACM **42** (1999) 32–48
19. Fehr, E., Gachter, S.: Altruistic punishment in humans. Nature (2002) 137–140

20. Schwartz, J., Tedeschi, B.: New software quietly diverts sales commissions. New York Times (2002) http://www.nytimes.com/2002/09/27/technology/27FREE.html.
21. Spring, T.: KaZaA sneakware stirs inside PCs. PC World (2002) http://www.cnn.com/2002/TECH/internet/05/07/kazaa.software.idg/index.html.
22. Weatherall, D.: Active network vision and reality: lessons from a capsule-based system. In: Proceedings of the Seventeenth ACM Symposium on Operating System Principles, Kiawah Island, SC (1999) 64–79
23. Hicks, M., Kakkar, P., Moore, J.T., Gunter, C.A., Nettles, S.: PLAN: A Packet Language for Active Networks. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages, ACM (1998) 86–93
24. Wallach, D.S., Balfanz, D., Dean, D., Felten, E.W.: Extensible security architectures for Java. In: Proceedings of the Sixteenth ACM Symposium on Operating System Principles, Saint-Malo, France (1997) 116–128
25. Reed, M.G., Syverson, P.F., Goldschlag, D.M.: Anonymous connections and onion routing. IEEE Journal on Selected Areas in Communication: Special Issue on Copyright and Privacy Protection **16** (1998)
26. Waldman, M., Rubin, A.D., Cranor, L.F.: Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In: Proc. 9th USENIX Security Symposium, Denver, Colorado (2000) 59–72
27. Waldman, M., Mazires, D.: Tangler: A censorship resistant publishing system based on document entanglements. In: 8th ACM Conference on Computer and Communcation Security (CCS-8), Philadelphia, Pennsylvania (2001)
28. Hazel, S., Wiley, B.: Achord: A variant of the Chord lookup service for use in censorship resistant peer-to-peer. In: Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, Massachusetts (2002)
29. Serjantov, A.: Anonymizing censorship resistant systems. In: Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, Massachusetts (2002)
30. Freedman, M.J., Sit, E., Cates, J., Morris, R.: Tarzan: A peer-to-peer anonymizing network layer. In: Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, Massachusetts (2002)
31. Dingledine, R., Freedman, M.J., Molnar, D.: Accountability measures for peer-to-peer systems. In: Peer-to-Peer: Harnessing the Power of Disruptive Technologies, O'Reilly and Associates (2000)
32. Bellovin, S.: Security aspects of Napster and Gnutella. In: 2001 Usenix Annual Technical Conference, Boston, Massachusetts (2001) Invited talk.