Protecting the PIPE from malicious peers

Brian F. Cooper, Mayank Bawa, Neil Daswani and Hector Garcia-Molina

Department of Computer Science
Stanford University
{cooperb,bawa,daswani,hector}@db.stanford.edu

Abstract

Digital materials can be protected from failures by replicating them at multiple autonomous, distributed sites. A significant challenge in such a distributed system is ensuring that documents are replicated and accessible despite malicious sites. Such sites may hinder the replication of documents in a variety of ways, including agreeing to store a copy but erasing it instead, refusing to serve a document, or serving an altered version of the document. We describe the design of a a Peer-to-peer Information Preservation and Exchange (PIPE) network: a distributed replication system that protects documents both from failures and from malicious nodes. We present the design of a PIPE system, discuss a threat model for malicious sites, and propose basic solutions for managing these malicious sites.

1 Introduction

For centuries, librarians and archivists have studied methods for preserving paper library materials. It is well understood how to handle issues such as acidic paper, humidity and broken bindings in the paper world. However, as more and more materials are made available digitally, it is clear that preservation must be re-thought in its entirety. Hard drives may crash, users may accidentally delete data, publishers may go out of business and shut down servers that are providing digital materials; these and a host of other vulnerabilities mean that a new infrastructure must be created to manage and preserve digital materials as effectively as paper materials are handled today. A basic solution to this problem is to make multiple copies of digital materials and spread them around to multiple libraries. Then, just like having paper copies of Charles Darwin's "On the Origin of Species" at multiple libraries all over the world, a failure at one library does not mean that the material is lost forever. Several systems that embody this replication concept for digital materials have been built, including OceanStore [13], LOCKSS [25] and SAV [5, 6].

In order for this approach to work, multiple libraries must cooperate to provide storage, searching and distribution for each digital item. These libraries can work together in a peer-to-peer (P2P) network, where every site both contributes and uses system resources. However, the very property that protects digital materials, namely storing copies at multiple distributed, autonomous sites, makes the P2P preservation system vulnerable to another problem: *malicious attacks*. For example, imagine that an attacker wants to prevent the distribution of an important document such as "Origin of Species." Such an attacker may be a fanatic opposed to the content of the book, similar to people who have used scissors to cut up books they disagree with in paper libraries. Or the attacker may be deranged and bent on destruction, like the man

who attacked Michelangelo's Pieta with a hammer. The attacker may just be a prankster, like the "script kiddies" that propagate viruses or trojans and conduct denial of service attacks in the Internet at large.

An attacker could set up a new site, or infiltrate an existing site, with the goal of subverting the preservation system. The attacker can make the site agree to store a backup copy of "Origin of Species," but delete the backup instead. If the primary copy is lost due to a failure, and the attacker had the only backup, then the document is lost permanently. Furthermore, the attacker can make the site claim to have a copy of "Origin of Species", but provide an altered version instead with key passages deleted or re-written. The attacker's site can agree to store a search index (like a digital card-catalog), but whenever it sees a search matching "Origin of Species," the site ignores the query or returns incorrect results. This makes it impossible to retrieve a copy of "Origin of Species," even if one exists.

There must be a system designed to prevent, detect, manage and/or recover from these malicious behaviors, so that the preservation infrastructure can provide services to its users. Certainly, a large amount of work has been done to address specific security issues in distributed information management systems. Examples of existing techniques include watchdog processes [19], byzantine agreement [17], distributed transaction commit [21], and cryptography. These techniques aim to preserve low-level system properties, such as distributed agreement, replicated data consistency, privacy, and authentication. However, in order to build a complete digital library, it is necessary to assemble these techniques into a system that provides end-to-end assurances. For example, replication techniques [10, 4] can be used to distribute copies of digital documents, but a complete system must also provide a secure, fault tolerant mechanism for performing content-based searches for digital documents. Similarly, a complete system must ensure that published documents remain preserved even if the publisher leaves the system, that users can verify the authenticity of retrieved documents, that agreements remain valid years after they are originally concluded, and so on. Furthermore, applying known security and reliability techniques in a straightforward fashion often leads to end-to-end solutions that are too expensive and inefficient. Not only do we need to develop new solutions that are more efficient, but we also need to understand and develop more "protection choices," e.g., solutions that may offer "slightly weaker" guarantees with respect to malicious sites, but are less costly.

Building a replication system on a peer-to-peer architecture has several advantages, including the fact that the resulting system has many more resources (in aggregate) than any one member. However, the challenge presented by malicious nodes is magnified by the loose coupling and autonomy of a peer-to-peer system. In particular, it is necessary to adapt existing techniques to work in a distributed, autonomous P2P architecture. For example, public-key cryptography [8] may be employed as a component of a preservation system. However, public-key techniques often require a centralized certificate authority, and this requirement must be reconciled with the P2P philosophy of "no centralized services". More related work is examined in Section 4.

In this paper, we discuss our vision of a reliable preservation service, which we call a *Peer-to-peer Information Preservation and Exchange* (PIPE) service. We are not presenting any complete solutions, but rather arguing that providing the end-to-end digital library services, despite failures and maliciousness, is a hard and unsolved problem, requiring the attention of our community. To make this argument:

- We present a strawman design for a PIPE system, listing the basic services that the system provides. This design provides a framework for discussion.
- We discuss the main challenges presented by malicious nodes to such a system.
- In order to deal with these challenges, multiple techniques must be used, and we sketch a basic im-

plementation for each of the PIPE services. This sketch will illustrate how existing techniques can be used to deal with the challenges in a PIPE system, and reveal the gaps that must be filled by ongoing research.

- Finally, we suggest techniques that can potentially fill these gaps.

While it may be impossible to *guarantee* safety from malicious nodes with any number of safeguards, our goal is to "raise the bar" to make it harder for nodes to act maliciously. As stated earlier, a deployed system must also be efficient and scalable. A PIPE network that places excessive resource requirements on its members is unusable, and an implemented system will have to embody techniques that are both secure and efficient.

This paper is organized as follows. In Section 2, we propose a PIPE architecture, sketch the services provided by the system, and identify the malicious behaviors that might occur. Then, in Section 3, we sketch implementations of the operations provided by the PIPE network, and identify key research questions for each operation. In Section 4, we examine related work, and in Section 5 we present our conclusions.

2 Basic PIPE architecture

The goal of a PIPE network is to preserve digital information and make it available to clients that request it. In this way, the PIPE infrastructure acts like a distributed, robust, electronic version of the "book stacks" of a traditional library: materials are stored in a controlled, protective environment, and served to users that need them. Other digital library services, such as user interfaces for accessing digital materials, rights management and user collaboration mechanisms can be built on top of the PIPE infrastructure to leverage its reliability and robustness. However, the PIPE network does not itself provide these functions; instead, it acts as a reliable substrate for higher level digital library services.

2.1 Peer-to-peer architecture

The PIPE network is composed of *peers*, or archive sites, that cooperate with each other to provide a preservation service. For example, a peer may be a university library, a government agency, or a corporation. This preservation service archives and serves *digital documents*. Examples of digital documents include JPEG images, Postscript documents, audio clips, or collections of scientific measurements. Because the peers are distributed, they must communicate by sending *messages* over some underlying communications network (e.g., the Internet).

Each peer provides resources for use by other peers. For example, a peer offers storage space for copies of digital documents, processing capability to answer searches, and bandwidth for serving documents. We chose a peer-to-peer architecture for the PIPE system because it fits well with the traditional model of libraries cooperating to provide a service (such as inter-library loan). Moreover, a peer-to-peer system is especially well suited for the problem of digital preservation because such a system can effectively leverage resources scattered at distributed, autonomous sites. First, the aggregate resources of the system (content, storage space, bandwidth, etc.) are larger than any one site has or can afford. Second, the system preserves the autonomy of sites, since each site makes local decisions when interacting with the network, rather than having to submit to a centralized controller. This makes it more likely that sites will participate in the system. Third, a peer-to-peer network is resilient to peer failures and localized network failures. This is because the network is a collection of distributed, heterogenous sites and binary communication links that

operate independently, even as other nodes and links fail. Finally, because backup copies of documents are stored "on-line" at other peers, lost or corrupted copies can be quickly restored simply by retrieving a copy of the backup over the network.

In our PIPE architecture, each node has a unique id. Each document must also have a unique id. This identifier is distinct from document metadata, such as title or author. This is because several documents may have the same title but should be distinguishable by the system. For example, it should be possible to tell the difference between "Origin of Species" (the classic work) and "Origin of Species" (the PBS special). Moreover, documents may have multiple copies, such that all copies with the same content have the same id. Node and document ids are used to facilitate unambiguous communication; asking for "document with id 543223" is more precise than asking for "Origin of Species". In Section 3.2 we discuss the challenges to deploying unique node and document ids in a network with malicious peers, and suggest techniques for securely implementing those ids.

2.2 PIPE services

The PIPE infrastructure should provide end-to-end preservation of digital documents. By "end-to-end preservation" we mean the system ensures that documents, once published, always exist in the system, can be discovered by users performing content-based searches, and can be retrieved by those users. More specifically, we propose a PIPE system that offers the following services:

I. *discover*(): Gives a new node i a list of nodes already in the PIPE network, and informs each of these existing nodes about i's existence.

Because the PIPE service is a collaborative effort between nodes communicating in a peer-to-peer manner, nodes must know about each other in order to communicate and collaborate. When a node first joins the PIPE network, it uses the *discover* operation to discover other nodes that have already joined and to announce its presence to these existing nodes.

II. *publish*(D, i): Given a document D by a publisher i, publishes D to the network.

The *publish* is the operation that ensures documents are both preserved and available for clients. Note that it is still possible for content creators to restrict access to their works according to their copyright. Digital rights management (DRM) techniques can be used to encrypt the bits so that only authorized users can make sense of them. The PIPE service operates at a lower level than the DRM, ensuring merely that the bits are protected from failures and that any peer can retrieve the bits, although those bits may be meaningless to unauthorized users due to DRM.

III. *search*(q): Given a descriptive search q (such as keywords or metadata), returns the ids of documents that match q, and the ids of nodes holding copies of those documents.

The *search* operation is necessary to locate content if the location is not already known. For example, a user may be searching for information about "evolution." Or, the user may be searching for the specific document "Origin of Species," but may not know its location.

IV. *retrieve*(D, i): Given a document id D and a peer id i, retrieves a copy of the document from the peer.

The *retrieve* operation is employed by users to get information that has been placed in the network via the *publish* operation. The user may perform a *search* to discover relevant documents, and once a promising document is selected, perform *retrieve* to get that document.

In Section 3, we sketch how various techniques can be employed to implement these operations in a secure manner.

2.3 Malicious node threat model

The framework of the previous section assumes that peers in the PIPE network cooperate to provide publishing, search and retrieval services. Unfortunately, a malicious node may appear to contribute to the service, but instead acts to subvert the service. Here, we examine malicious activities (the *threat model*) that are unique to a PIPE system. If these behaviors are surreptitious, it may be difficult to discern that the node is malicious or specifically what its malicious behavior is.

A. Publish a document using the same id as an existing document.

This makes it difficult for a user to find authentic documents. For example, the user may find (via *search*) that the id of "Origin of Species" is D. When the user goes to retrieve D, he may instead get an altered version with the same id produced by a malicious node.

B. Agree to store a copy of a document, but delete it instead.

If a malicious node deletes a document that the node is supposed to be storing, it is difficult to ensure that the document is preserved. Imagine that three copies of a document are published, but that two are stored at malicious nodes who then delete them. Because three copies ostensibly exist, the publisher may not make any more copies. When the remaining copy is lost due to failure, the document will be lost permanently.

C. Agree to store a copy of a document, but refuse to perform *search* operations over the document.

D. Agree to store a search index for documents, but refuse to perform (some or all) *search* operations.

Even if a document is preserved, if users cannot find it, it is effectively lost. A malicious node who refuses to perform search operations for a document effectively decreases or eliminates the availability of that document

E. Agree to serve a document, but serve an altered copy instead, or refuse to serve the document altogether.

Again, even if a document exists, it must be retrievable to be useful. If a malicious node refuses to serve the document, it is effectively lost. Moreover, if the malicious node serves an altered version, the original is still effectively lost.

F. Coordinate attacks with other malicious nodes.

Some malicious behaviors are easy to detect or mask if malicious nodes act alone. For example, if there are four copies of a document, and two are altered by malicious nodes, it is relatively easy to determine which two copies are authentic, since those will be the only two copies that are identical. However, if the malicious nodes work together, it will be difficult to tell from the documents themselves which two copies are altered and which two copies are original.

G. Masquerade as a different peer.

H. Modify otherwise authentic messages.

Because a PIPE network requires cooperation between peers, effective communication is necessary as part of this cooperation. A malicious node may attempt to disrupt a service by falsifying system messages. For example, a publisher node may want to verify that at least three copies of its published documents exist in the network. If a malicious node can masquerade as other nodes or falsify response messages, the malicious node can convince the publisher that multiple nodes hold a copy when in fact nobody does. Then, the publisher may fail to make necessary extra copies, and if the publisher itself fails, the document will be lost.

I. Lie in response to any request for information.

Sometimes it is necessary request information from nodes in order to implement certain operations. For example, a new node i may wish to discover other nodes in the network, and may send a request to some node j asking for a list of other nodes. If j is malicious, it can lie, claiming that j and its malicious confederates are the only nodes in the network. Thus, while there may be good nodes willing to collaborate to preserve i's information, i will never be able to learn of their existence or contact them, and will be unable to effectively take advantage of the PIPE services to achieve preservation.

In addition to these attacks, a PIPE network is also vulnerable to malicious activities common to any distributed system, such as viruses, trojans, hackers, denial of service attacks [22], and so on. Much research is currently focused on these challenges. Techniques that are useful for general distributed systems are also useful for a PIPE network. Here, we focus on the problems unique to the PIPE network.

3 Implementing a PIPE service

The basic model of Section 2 defines the operations that must be implemented by a PIPE network, and the obstacles to implementing those services due to malicious nodes. In this section, we examine two scenarios: a relatively closed network, where the main challenge is failures, and a more open network, where both failures and malicious nodes are issues. Here we will sketch a "strawman" implementation of the PIPE service for these scenarios, and outline key research questions that must be answered.

3.1 Model F: failures

We first examine how a PIPE network manages failures (as opposed to malicious attacks). Of course, there is nothing surprising in our solutions (failure handling is well understood), but it does serve as a solid starting point for dealing with malicious sites. Moreover, in some cases, failures, not malicious nodes, may be the main threat. For example, a PIPE service may be deployed in a closed environment, such as within a single enterprise. While the nodes of the PIPE system may be geographically scattered at the various corporate offices, the enterprise can retain control of the peer machines, software and data storage. Although the occasional disgruntled employee or hacker may penetrate the system, the enterprise can deploy the necessary precautions (monitoring software, firewalls, etc.) to reduce the risk of malicious attacks.

We assume that failures are fail-stop; that is, a node fails by ceasing to operate (but does not act maliciously). It does not respond to messages or generate new messages, and may lose data. A *live* node is a node that has not failed. Nodes may experience failures for a variety of reasons, such as hardware faults, software bugs, natural disasters, bankruptcy, and so on.

The main challenge in this scenario is to ensure that data is not lost due to a failure, and the basic solution is to replicate the data. Assume that for a given network of n nodes, we know that at least $n - k$ will always be live. At some point, a node that has failed may begin to operate again, and rejoin the network. However, this peer may have lost some or all of its information.

Although it is impossible to determine k exactly, we may be able to estimate it, for example by estimating the probability of failure of each node. Estimating k illustrates a tradeoff between being conservative and being efficient. Decreasing k reduces the resource requirements of the system but also decreases the fault tolerance. One key research goal is to develop techniques for effectively estimating k and therefore the number of copies that must be made.

The first step towards establishing a PIPE replication network is for nodes to learn about each other's existence so that they can communicate. This process occurs incrementally whenever a new node i joins the network:

- *discover*(): Discover $k + 1$ nodes via an out of band mechanism. Contact at least one of them and ask for a list of nodes in the network. Contact each of the nodes in the network and announce the new node's existence.

The *discover* operation requires the new node i to know the identity of at least one node j already in the PIPE network in order to bootstrap the process of learning the identities of all nodes in the network. Because there may be k failures, $k + 1$ nodes may have to be tried before a live node is found. The $k + 1$ existing nodes can be discovered using a centralized list of server ids, an anycast service, emails from administrators of sites in the PIPE network, or some other out-of-band mechanism. When i contacts an existing node j, j learns of the new node i's existence, and gives a list of other nodes in the network to i. Once i contacts these other nodes, all nodes will know of i's existence. Unfortunately, the need for each node to know about all other nodes in the system may limit scalability. More efficient mechanisms may be developed to discover nodes dynamically as necessary. We present this implementation as a starting point.

Since up to k nodes may fail, the *publish* operation should guarantee that the document is preserved despite these failures. One basic solution is to make multiple copies of documents:

- *publish*(D, i): Store at least $k + 1$ copies of the document D, each copy at a different peer (including i).

Even if there are k copies lost due to failures, there will still be one good copy available (at some live node). Various techniques can be used for selecting which peers are given a copy (see for example [6]).

However, once a node fails, we need to take steps to make more backups of the document, since that node may lose data or never rejoin the network. To do this, there must be two additional operations:

- *detect-failure*(D, i): Probe node i to ensure that it is live and has a copy of D.
- *recover*(D): Make additional copies of D until there are at least $k + 1$ copies at live nodes.

Each node that has a copy of document D should periodically perform *detect-failure* to probe all other nodes holding a copy of D. The probe could be as simple as asking for a checksum or CRC of the document. As soon as any node notices that another peer has failed or a document is lost, it can use *recover* to ensure that there are still at least $k + 1$ copies. *Recover* is a lightweight operation, unlike traditional database recovery mechanisms where complex transactions must be undone or redone. Note that because of *detect-failure* there

is no need for an explicit "goodbye" operation when a node leaves the network. Other nodes that are still in the network will notice that the node has left. It may be useful to implement some sort of "goodbye" protocol to speed up the process of detecting when a node has voluntarily network, but such a protocol is not necessary for correctness and can be omitted for simplicity.

The *search* operation must ensure that digital documents can be found, even if the original publisher (who knew the location of all the copies) has failed.

- *search*(q): Broadcast the query q to all peers. Broadcast can be performed by sending an identical search message to all peers. If a peer i has a document matching the search, that peer returns the id D of the document as well as i, its own id.

This implementation will certainly allow nodes to locate documents. However, the use of broadcast also limits the scalability of the system. More efficient techniques should be deployed to ensure both robustness in the face of failures and efficiency. For example, the PIPE system can adopt a supernode architecture, such as that in Kazaa's FastTrack protocol [16]. Each supernode indexes the content held by other, normal nodes. In this architecture, only a few nodes need to handle searches, increasing scalability. However, it is necessary to replicate index information at multiple supernodes so that searches can be handled despite a supernode failure.

Once a document's location has been determined, the *retrieve* operation can be implemented in a straightforward manner:

- *retrieve*(D, i): A message containing the document id D is sent to the peer i. That peer returns a copy of the document.

3.2 Model FM: failures and malicious nodes

A network of autonomous, cooperating institutions faces more challenges than the closed environment described in the previous section. In this situation, we must deal with the possibility that nodes may be malicious. A malicious node may continue to receive, process and send messages, but may do so in order to subvert the network for its own ends. A *good* node lives up to its agreements and responsibilities (although up to k good nodes may fail, as in Model F). A *malicious* node does not live up to these responsibilities. The activities that may be undertaken by a malicious node are laid out in the threat model of Section 2.3. In Model FM, we assume that in a network of n nodes, up to m nodes may be malicious at any one time. Recall that up to k nodes may also simultaneously fail.

3.2.1 Peer communication

In a PIPE system, nodes must be able to effectively communicate in order to cooperate. Two components are necessary to ensure effective communication despite malicious nodes: secure, unique peer ids and secure communication channels. In fact, we argue that these two components are sufficient as well as necessary for building a PIPE network that is robust in the face of malicious nodes. Without both secure peer ids and secure channels, effective peer-to-peer communication cannot be guaranteed, but with these components, we have a complete communication infrastructure for deploying our basic techniques (described in Section 3.2.2).

The first component that is needed is **secure, unique peer ids**. A malicious peer may try to masquerade as another peer, but this behavior should be detectable by "good" (non-malicious) nodes. Peer authentication is vital to many operations in the PIPE network. For example, it is vital that nodes accepting a copy of

Figure 1: Network topologies.

a document as part of a *publish* operation be able to verify that the node sending the copy is in fact the publisher and not some malicious node peddling a forgery. Similarly, if node A requests a document from node B via *retrieve*, it is necessary to verify that the returned document came from B and not some other node. Node ids can be securely attached to nodes in a variety of ways. One way is to generate a public key-private key pair, and use the public key as the node id. In this scheme, a node "signs" (encrypts) all messages with its private key. An alternate scheme is to have some authentication authority that can verify node ids. Unfortunately, both alternatives require centralized certificate or authentication authorities. Research may be necessary to develop techniques that provide secure, unique peer ids with minimal dependance on a central authority. Ideally, no central authority would be needed, since such an authority could be a single point of failure or security vulnerability.

The second component, **secure communications channels**, ensures that messages arrive at their destination. By *secure* we mean that a peer i should be able to send messages to any other peer j that 1. are authenticated as coming from i, 2. are unmodified and 3. arrive or detectably fail to arrive at j (so that i can resend if necessary). We do not require that messages be private (e.g., unreadable by third parties). Because our techniques rely on peers communicating with each other, it is vital that peers be able to send messages to each other, despite the presence of malicious nodes. Secure channels can be implemented by known, reliable transmission protocols (such as secure HTTP) although these protocols may require a public-key infrastructure. An alternative is to use dedicated physical channels. In other words, if peer i had a direct physical connection to peer j, then i would be able to send messages securely to j, and vice versa.

In some PIPE networks, it may be difficult to ensure secure channels. For example, some networks (such as Freenet [11] or systems based on Gnutella [15]) are *partially connected*, such that each peer only communicates with a limited set of neighbors. An example of a partially connected network is shown in Figure 1(a). In such a network, peers are expected to *forward* messages in addition to responding to them. A malicious node may be able to disrupt secure channels by refusing to forward messages. In contrast, in a *fully connected* network, every peer can communicate directly with every other peer; an example is shown in Figure 1(b). Research is necessary to develop techniques for secure communication channels over partially connected networks.

Other techniques must also be employed to prevent or mitigate attacks that are common to any distributed system. For example, denial of service attacks can be used to disrupt peer communication. As noted in Section 2.3, such attacks are not unique to a PIPE system. Techniques that are applicable to generalized distributed systems can be used for PIPE networks as well.

3.2.2 PIPE operations

The operations of Section 2.2 must be implemented in a way that is both fault-tolerant and impervious to malicious attacks. In this section we sketch solutions that work but are not necessarily efficient. (In

Section 3.3 we discuss efficiency.)

First, the *discover* mechanism must be able to tolerate malicious nodes:

- *discover*(): Discover $k + m + 1$ nodes via an out of band mechanism. Contact at least $m + 1$ of them and ask for a list of nodes in the network. Contact each of the nodes in the network and announce the new node's existence.

With the implementation given in Section 3.1, a new node i contacts one other node j. However, if node j is malicious, it could return a list of either malicious nodes or invalid node ids, so that i would not have a valid list of peers. If i contacts $k+m+1$ sites, it will at least get one list from a good, active node, although it may get up to m lists from malicious sites. Since i does not know which is the good list, and does not know which are invalid node ids (the nodes could simply be down temporarily), node i unions all the lists to obtain its list of nodes. Node i must retain even ids that are seemingly invalid, since these may be valid nodes that rejoin the network at some point in the future.

Next, the *publish* operation must guarantee that at least one copy of document is preserved at a good, live node, despite k failed nodes and m malicious nodes.

- *publish*(D, i): Store $k + m + 1$ copies of document D, each copy at a different peer (including i).

Malicious nodes may delete documents, refuse to serve documents, or serve the wrong document, but there will still be a good document being served *somewhere*.

This approach presents two challenges:

- Determining m, the maximum number of malicious nodes
- Determining when a document is stored at a malicious node

Both problems are quite difficult, since in practice it is impossible to detect all malicious nodes. For example, if node X is behaving badly (for example, refusing to serve documents), that is easy to detect, since *retrieve* requests are always refused. However, imagine a node Y that is waiting for the right time to strike. For most of its lifetime, it acts "good," responding to requests and serving documents. However, at some point Y may notice that there have been failures in the network, and that it now holds the last copy of some document D. If Y then deletes D, that document is lost forever. Such behavior is impossible to predict beforehand.

Therefore, a key research question is to determine which malicious acts can be detected. Undetectable maliciousness must be masked by extra redundancy, so that even if a node becomes unexpectedly "evil" there are enough "good" copies of documents so that the evil behavior is not detrimental. Research is also needed to balance extra redundancy with the resource limitations of the system.

In addition, we can raise the barrier to maliciousness somewhat by defining operations to detect some of the more obvious malicious behaviors. For example:

- *detect-servecopy*: Ask a node to serve a document it is supposed to store. If the node refuses, then it is acting maliciously.
- *detect-has-copy*: Choose a random portion of a document that a node is supposed to be storing, and ask the node to return that portion. If the node cannot, or the returned portion does not have the expected content, it has destroyed the document and is therefore malicious.

Each node h should take the responsibility of probing remote nodes that store copies of the same documents as h. If any detection operation indicates that a node is acting in bad faith, then extra copies of documents held by the malicious node can be made to better protect the document. The *recover* operation can be used for this purpose.

When using *recover* after a failure or malicious behavior is detected, nodes must be careful about which copy is chosen as the "good" document to replicate. If an altered or incorrect version is inadvertantly chosen, then the malicious behavior will be enhanced rather than mitigated.

The *search* operation can be implemented as before, by broadcasting the query to all peers. This guarantees that any good, live documents which match the search will be returned to the user. However, in addition to "correct" search results, a malicious node may return bad results which must be filtered out by the searching node. Part of the difficulty in this process is the ambiguity in distinguishing between "junk" and "real documents." For example, imagine that a user wants to find the book "Origin of Species." There may be multiple documents titled "Origin of Species," including the classic Darwin book and also other books written by other people that are not necessarily acting maliciously. Moreover, specifying more metadata (such as the author or year) may not help. For example, a PhD thesis titled "Origin of Species by Charles Darwin in 1859" will "legitimately" match even detailed searches. In contrast, some documents are clearly "junk." If a malicious node rewrites "Origin of Species" in order to deliberately deceive readers, then the altered version is certainly junk.

An important research problem is to develop techniques for filtering out this junk. This is not an easy problem; how would a user that has never read "Origin of Species" know which document was real and which had been altered? One possibility is for some authority (such as the Library of Congress) to serve as a document authenticator. This solution unfortunately decreases the robustness of the system, since the authority becomes a single point of failure. Another potential technique is to use *secure timestamps* [20] to distinguish between documents based on creation time. While secure timestamps can be implemented in a distributed way, they may not offer enough information to properly filter junk.

Finally, the PIPE network must securely implement *retrieve* to guarantee that a good, live copy of any document published to the PIPE system can be retrieved by the user.

- *retrieve*(D, i): A message containing the document id D is sent to the peer i, and i returns a copy of the document.

A malicious node may produce a fake or altered document in response to a *retrieve*. To check the success of *retrieve*, user can verify that the document matches the requested id. For example, a user may perform *retrieve* for a copy of "Origin of Species." If the retrieved document matches the request id, then the user knows that the retrieved document is an authentic, unmodified copy of Darwin's classic work. If not, the *retrieve* operation has failed and must be repeated for a different node that claims to have a copy of the document.

The *retrieve* operation therefore requires that document ids must be securely bound to documents. Secure document ids will also help with the *recover* operation, where nodes must verify the authenticity of a document before making new copies. One way to securely bind document ids to documents is to calculate the id from the document content; such an id is called a *signature*. For example, an algorithm such as SHA1 or MD5 can be used to calculate a signature in a secure way. Since the signatures are shorter than the documents themselves, it is possible that multiple documents could produce the same signature; however, such an occurrence is so extremely improbable as to be effectively impossible. With a signature-based scheme,

it is easy to verify that a retrieved document matches the requested id simply by recalculating the signature over the retrieved document and comparing it to the id.

3.3 Improving on Model FM's operations

In Section 3.2 we sketched some simple techniques for ensuring preservation despite malicious nodes. These simple techniques are "brute-force;" they rely on broadcast and a high degree of replication. A key research challenge is to develop techniques that more efficiently use resources such as bandwidth or storage while still ensuring the basic preservation properties of the system. In this section we propose starting points for developing techniques that meet the requirements of both efficiency and security.

3.3.1 Improving detection of malicious nodes

The *detect-servecopy* and *detect-has-copy* operations attempt to probe a site to see if it is operating correctly. Although this probing may detect some malicious behaviors, others may occur unnoticed. Moreover, probing requires many messages to be sent, using up network bandwidth, even when no sites are malicious. If peers are participating in the system because they want to derive benefit from it, one alternative is to use *incentive-based* verification. In this approach, nodes only derive benefit from the system if they behave correctly.

One example of incentive-based methods is to make nodes prove they are storing documents that they have agreed to store in order to retrieve other documents. Imagine that a node M is responsible for storing a copy of a document D_1. M may perform a *retrieve* operation for another document D_2 stored at P. If P knows that M is supposed to be storing D_1, and is itself storing a copy of D_1, then P can send $D_3 = D_1$ XOR D_2 in response to the *retrieve*. M can then "decode" D_2 by performing D_3 XOR D_1. M has an incentive to store D_1 despite otherwise malicious inclinations; otherwise, it cannot read retrieved documents. This scheme effectively spreads a detection operation over the *publish* and *retrieve* operations: *publish* must distribute a list of who should be storing what so that the correct test can be performed at *retrieve* time. Unfortunately, this scheme does not ensure that a node will actually serve a document that it is storing. Moreover, if P does not have any documents in common with M, then it cannot perform the challenge at all. At the same time, a malicious node may join the network simply for the purpose of subverting the system, and may never itself issue a *retrieve* operation. More research is necessary to develop this scheme to serve as the basis of a robust incentive-based mechanism.

3.3.2 Improving *search*

Clearly, broadcast is an inefficient mechanism for implementing the *search* operation. A better alternative is to employ indexing. Indexes, such as inverted lists and other structures developed for information retrieval, can be used to implement the *search* functionality.

Conceptually, the *search* operation has two purposes: to learn which documents match a particular *query*, and to *locate* peers that have a copy of those documents. The requirements for indexing are different for each of these tasks. The *query* task is similar to information retrieval, where searches are conducted over the text or metadata of the document, and the search result indicates the "quality" of the match. In contrast, the *locate* task is a determination of which nodes have a particular document. This does not require the text or the metadata, but only the id of the document. Similarly, there is no notion of "quality" for a *locate* result, but only "yes" or "no" for each site. A dictionary, such as a hash table, can be used for *locate*.

Therefore, it makes sense to decompose the *search* operation into two operations: *query*(q) and *locate*(D). The *query* operation takes the search terms q over the document or its metadata, and returns a set of document ids that match the query, possibly along with a "quality" rating for the match and/or metadata about each match. The *locate* operation takes a document id D and returns a set of peers that have a copy of the document. Finally, the document can be obtained from one of these peers using *retrieve* (as before).

Another advantage of having separate *query* and *locate* operations is that each can be implemented and optimized separately. For example, imagine a network where documents, once published, move frequently from node to node. This may be the case if there is a high rate of failure among nodes, and new copies of documents are continually made as old copies are lost. In this case, it is necessary that the *locate* index is updated frequently. However, the *query* index does not need to change in response to node failures, since the documents matching a search do not change even as the physical location of copies of those documents changes.

The indexes for *query* and *locate* must be located at nodes in the PIPE network (although it is not necessary that the *query* index and *locate* index reside at the same peer). In the simplest case, every node can have a copy of both indexes, but then both the resource requirements at each node and the update cost for each index would be prohibitive. One better alternative is to have indexes at *supernodes*: special nodes responsible for indexing. Then, there are fewer indexes to update and only the supernodes must expend the storage resources to store the index, and the bandwidth and processing resources to answer queries. This alternative has the disadvantage that the supernodes may fail or be malicious. In this case, there must be multiple copies of each index. In other words, the index is conceptually replicated just like a regular document, although it is really the indexing process, not the contents, that are being replicated.

As before, "junk" search results can be filtered by using secure timestamps or an authority like the library of Congress. There must be $k + m + 1$ different nodes that have each type of index so that there is at least one live, uncorrupted index. A peer performing *query* or *locate* can send lookup messages to index nodes until a good, live index is found. In the worst case, this will require $k + m + 1$ lookups. If there are more indexes, we can discover from the indexes themselves which are the correct results. If there are $k + m \times 2 + 1$ indexes, then there will be at least $m + 1$ good, live indexes. A peer can query all live indexes, and only accept search results that appear in at least $m + 1$ indexes. This approach has the disadvantage that more indexes must be queried for every search, but reduces the need to rely on timestamps or an authority to discover good search results.

Once index locations are determined, they can be built incrementally as documents are published and copied. A *publish* of a document should result in an update to each *query* index for the contents of the document, and an update to each *locate* index for the locations of the copies. Whenever a new copy is made (e.g. by *recover*) each *locate* index must be updated. If an index is lost due to peer failure, then another indexing node must perform an *index recover* to make a new copy of the index. It is not sufficient to simply copy an index from a live node; that node may be malicious and the index may be deliberately corrupted. As with *query* and *locate*, each index entry can be verified using secure timestamps, a central authority, or by querying all indexes and accepting entries that appear at least $m + 1$ times.

After indexes are built, their location must be made known to other peers, so that those peers can conduct searches. The simplest mechanism is that a node notifies the network, using broadcast, that it has an index. This reduces the use of broadcast to a one-time setup message.

4 Related work

Replication to protect against failures has been employed in several systems, such as RAID [23], mirrored disks [3], replicated file systems [10], and so on. In most of these systems, all of the replicas are under the control of a central authority, and thus the issues of dealing with autonomy and maliciousness are less relevant.

A variety of techniques have been investigated for protecting against malicious activities in a distributed system. One class of techniques are based on cryptography, including secret key [1], public key [8] and cryptographic hashing [2] mechanisms. If nodes fail and lose their keys, key escrow techniques [7] can be used to recover. However, many protocols based on cryptography assume a central authority or set of authorities, for example to issue certificates. This assumption may not be appropriate in a distributed, autonomous system, especially if the authority itself may fail. Moreover, it is not clear how to use encryption technology to prevent a malicious node from failing to store a document, or to deal with similar problems that we examine here. Another class of techniques seek to specify security in a declarative way. Systems such as PoET [24] can be used to enforce access control over digital objects, but do not address the problem of ensuring an object is stored and served.

Much recent research has focused on peer-to-peer systems. Several systems have been developed, including academic projects such as Chord [12] and Pastry [26] and industrial systems such as as Limewire [18] and Kazaa [16]. Much of this work has been focused on efficient and scalable search (as in Chord) or on publisher and searcher anonymity (as in FreeHaven [9]), rather than on the malicious behaviors we examine here. Peer-to-peer systems such as LOCKSS [25] and Archival Intermemory [14] deal with malicious nodes by making lots of extra copies, similar to what we propose in our basic techniques. Our PIPE system builds on these earlier systems by integrating search and publishing into one fault-tolerant, secure system.

5 Conclusion

Peer-to-Peer Information Preservation and Exchange (PIPE) networks are the central component of a robust, community-based digital library. In this paper we have outlined the PIPE service, including the operations it provides, the peer-to-peer architecture of cooperating nodes, and the properties (such as document preservation) it aims to achieve. We have also sketched how the system could be deployed to handle failures and malicious activities.

Examining the possible attacks, it is clear that digital libraries are quite vulnerable to malicious attacks. One could say that they are significantly more vulnerable than conventional libraries, where it is often easier to control who comes in contact with the library materials. In the digital world, any teenager with a PC can try to impersonate library servers, can inject into the system fake and altered documents, or can mount denial-of-service attacks.

We have also seen that protecting against malicious attacks is inherently costly. For every possible malicious site, we need to make extra copies of our documents. Every time we handle information, we have to be cautious: where did it come from? do several sites agree with this? when was this actually created? Even though there are potential optimizations, the bottom line is that it is still very expensive to protect against malicious users, a fact we may just have to live with.

Given the high cost of the mechanisms we have presented, it is clear that much research must be done to develop more efficient techniques. For example, it may be tempting to consider another class of "prob-

abilistic" mechanisms. Instead of making a fixed number of copies of a document (e.g., $k + m + 1$), sites would simply strive to make "lots" of copies of all documents. For example, a site may cache copies of say a random 5% of the documents it sees from other sites. The hypothesis is that, since each document has many copies, and no one knows exactly where they are, most documents will be protected. We believe that such an approach requires additional study, but in the end we suspect that many more copies of documents will be needed, and that there will be a danger that some documents (perhaps the less popular ones) will be lost.

References

[1] FIPS PUB 197. Advanced encryption standard, 2001. *Federal Information Processing Standards Publication*, U.S. Department of Commerce, National Bureau of Standards, Springfield (Virginia).

[2] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk. Cryptographic hash functions: A survey, July 1995. Technical Report 95-09, Department of Computer Science, University of Wollongong.

[3] A. Borr. Transaction monitoring in Encompass [TM]: Reliable distributed transaction processing. In *Proc. 7th VLDB*, Sept. 1981.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*, 1999.

[5] B. Cooper, A. Crespo, and H. Garcia-Molina. Implementing a reliable digital object archive. In *Proc. European Conf. on Digital Libraries (ECDL)*, 2000.

[6] B. Cooper and H. Garcia-Molina. Creating trading networks of digital archives. In *ACM/IEEE Joint Conference on Digital Libraries, Roanoke, VA, June 2001*, pages 353–362, 2001.

[7] D.E. Denning and D.K. Branstad. A taxonomy for key escrow encryption systems. *Communications of the ACM*, 39(3):34–40, 1996.

[8] W. Diffie. The first ten years in public key cryptography. *Proc. of the IEEE*, 76(5):560–577, 1988.

[9] R. Dingledine, M.J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 67–95, 2000.

[10] B. Liskov et al. Replication in the Harp file system. In *Proc. 13th SOSP*, Oct. 1991.

[11] I. Clarke et al. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[12] I. Stoica et al. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM*, 2001.

[13] John Kubiatowicz et al. OceanStore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, Nov. 2000.

[14] Y. Chen et al. A prototype implementation of archival intermemory. In *Proc. of the ACM Conf. on Digital Libraries*, 1999.

[15] Gnutella protocol specification. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.

[16] Kazaa home page. http://www.kazaa.com/.

[17] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. ACM Transactions on Programming Languages and Systems, Vol.4, No.3, pp. 382-401, July 1982.

[18] Limewire.org web site. http://www.limewire.org.

[19] A. Mahmood and E.J. McCluskey. Concurrent error detection using watchdog processors - a survey. *IEEE Trans. Comput.*, 37(2):160–174, Feb. 1988.

[20] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proc. of the 11th USENIX Security Symposium*, Aug. 2002.

[21] C. Mohan, R. Strong, and S. Finkelstein. Methods for distributed transaction commit and recovery using byzantine agreement within clusters of processors. In Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing, 1987.

[22] D. Moore, G. Voelker, and S. Savage. Inferring internet denial of service activity. In *Proc. of 2001 USENIX Security Symposium*, August 2001.

[23] D. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Record*, 17(3):109–116, September 1988.

[24] S. Payette and C. Lagoze. Policy-carrying, policy-enforcing digital objects. In *Proc. European Conf. on Digital Libraries (ECDL)*, 2000.

[25] V. Reich and D. Rosenthal. Lockss (lots of copies keep stuff safe). Preservation 2000, Nov. 2000.

[26] A. Rowstron, P. Druschel, and P. Scalable. Distributed object location and routing for largescale peer-to-peer systems. In Proc. IFIP/ACM Middleware 2001, Heidelberg, Germany, November 2001.