Samsara: Honor Among Thieves in Peer-to-Peer Storage

Landon P. Cox
lpcox@umich.edu

Brian D. Noble
bnoble@umich.edu

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

ABSTRACT

Peer-to-peer storage systems assume that their users consume resources in proportion to their contribution. Unfortunately, users are unlikely to do this without some enforcement mechanism. Prior solutions to this problem require centralized infrastructure, constraints on data placement, or ongoing administrative costs. All of these run counter to the design philosophy of peer-to-peer systems.

Samsara enforces fairness in peer-to-peer storage systems without requiring trusted third parties, symmetric storage relationships, monetary payment, or certified identities. Each peer that requests storage of another must agree to hold a *claim* in return—a placeholder that accounts for available space. After an exchange, each partner checks the other to ensure faithfulness. Samsara punishes unresponsive nodes *probabilistically*. Because objects are replicated, nodes with transient failures are unlikely to suffer data loss, unlike those that are dishonest or chronically unavailable. Claim storage overhead can be reduced when necessary by forwarding among chains of nodes, and eliminated when cycles are created. Forwarding chains increase the risk of exposure to failure, but such risk is modest under reasonable assumptions of utilization and simultaneous, persistent failure.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; K.6.4 [**Management of Computing and Information Systems**]: System Management

General Terms

Management, Performance

Keywords

Peer-to-peer storage systems, distributed accounting

1. INTRODUCTION

A peer-to-peer storage system offers a number of significant advantages. Each node stores its data on some remote hosts, and agrees to accept data from other hosts in return. Each object is replicated across an independent set of remote nodes in the collective, providing strong fault-tolerance and survivability in the face of node failure. Nodes can enter and leave the collective without administrative action; such systems are entirely decentralized, self-organizing, and scalable [2, 7, 8, 23].

These systems depend on individual nodes consuming storage space in proportion to their contribution. Without such balance, the demand for storage space exceeds supply, and the system collapses. Because the benefit of using the system accrues individually, but the costs are shared, users have little incentive to contribute beyond goodwill. This is called the *tragedy of the commons* [15]. Recent studies of Gnutella and Napster confirm that many users consume without contributing [1, 24]. Worse, some users under-report their available resources to avoid providing them to others [24].

Several mechanisms to compel storage fairness have been proposed, but all of them rely on one or more features that run counter to the goals of peer-to-peer storage systems. Trusted third parties can enforce quotas and certify the rights to consume storage [23] but require centralized administration and a common domain of control. One can use currency to track the provision and consumption of storage space [16], but this requires a trusted clearance infrastructure. Finally, certified identities and public keys can be used to provide evidence of storage consumption [16, 21, 23], but require a trusted means of certification. All of these mechanisms require some notion of centralized, administrative overhead—precisely the costs that peer-to-peer systems are meant to avoid.

If storage relationships in peer-to-peer systems are always symmetric, then the problem of enforcing fairness is much simpler. In a symmetric storage system, node A stores data on node B if and only if B also stores data on A. In such a system, B can periodically check to see if its data is still held by A, and vice versa. Collectively, these pairwise checks ensure that each node contributes as it consumes, and some systems require symmetry for exactly this reason [6, 18].

Unfortunately, symmetry is rare in most peer-to-peer systems. In some, an underlying distributed hash table [22, 26] directs replica placement. Nodes are given random, numerical identifiers within an ID space, and replicas of a data object are hashed and stored on the nodes with the numerically closest identifiers. Other systems provide more freedom in the choice of replica sites [2, 7], but such decisions are influenced by factors such as node locality, diversity, content overlap, and availability. Requiring symmetric storage in such systems constrains data placement unnecessarily.

(a) A simple data storage network

(b) One claim per stored block

(c) With claim forwarding
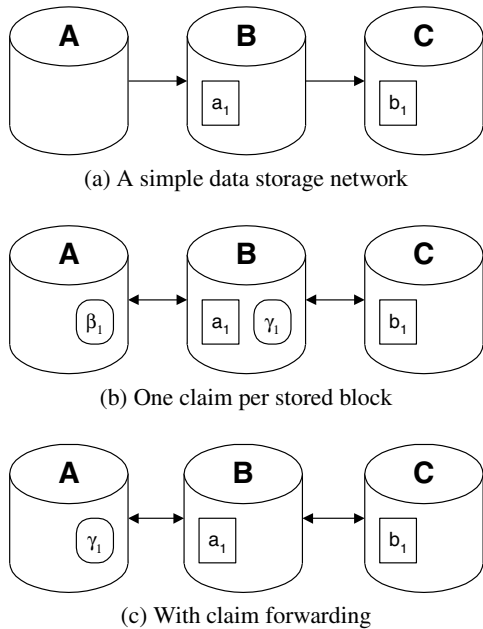
This figure illustrates a simple data storage network. The top panel shows only the blocks stored by each node. Node B stores block a_1 for node A, and C stores b_1 for B. The middle panel shows the network with the addition of storage claims. Node B must store claim γ_1 for node C, and A must store β_1 for B. In the bottom panel, node B forwards claim γ_1, rather than generating its own. This reduces storage overhead, but creates a dependency chain; B's data now depends on the faithfulness of A.

Figure 1: Storage claims

In this paper, we present Samsara, an infrastructure for enforcing fairness in peer-to-peer storage systems. To the best of our knowledge, Samsara is the only system that enforces storage fairness without requiring trusted third parties, monetary payment, certified identities, or symmetric storage relationships.

Samsara is based on the following observation: symmetric storage relationships can be *manufactured* where they do not arise naturally. For example, consider the asymmetric storage network in Figure 1(a). There are two blocks stored in this system. Node A stores one block on B and B stores a block on C. Samsara manufactures symmetry through the construction of *storage claims*—incompressible placeholders for storage. Each stored block represents the transfer of storage rights; one node contributes those rights and another consumes them. For example, C contributes storage that is consumed by B, and B contributes storage consumed by A. In Samsara, each contributing node creates a claim that the corresponding consuming node must store, as shown in Figure 1(b).

An exchange of data for claim forms a symmetric storage contract; each node periodically checks the other to ensure that it is adhering to the contract. If a node breaches the contract, its exchange partner is free to drop its data. These periodic inspections are done for purely selfish reasons; each node is concerned only with the maintenance of its own data and claims, and not with the system as a whole. Collectively, this ensures that all nodes act fairly—each node provides at least as much storage as it consumes.

This simple scheme leads to three important questions. How can the storage overhead of this scheme be reduced as utilization increases? When a node is found to be cheating, how much data must be discarded? When a node does not respond to a query, has it suffered a transient failure, or is it cheating?

Storage claims require space overhead equal to the data stored in the system. As space becomes scarce, claim overhead can be reduced or even eliminated through *forwarding*. When node A stores data on node B, we say A is *downstream* of B; this is a transitive relationship. Nodes are free to forward a claim downstream, rather than manufacture a new claim. For example, in Figure 1(c) node B is midstream of nodes A and C. If space on B is scarce, it can forward the claim γ_1 to A, replacing the claim β_1 and reducing local storage overhead. Note that B must still contribute at least one block in exchange for consuming space via b_1. However, this block is occupied by the data a_1 rather than the claim γ_1.

A node is still responsible for claims it has forwarded. In our example, B is still responsible for the claim γ_1, even though it is stored on A. If A cheats, B will be penalized. because it is ultimately responsible for γ_1. In other words, claim forwarding can reduce the reliability of data. Thus, nodes in Samsara retain claims rather than forward them whenever possible—individual users have both the incentive and the means to increase the reliability of their data stored elsewhere by contributing more storage to the system.

Peer-to-peer systems must tolerate occasional, transient failure of their constituents. Therefore, data stored in such systems is fact to ensure high availability. Samsara leverages this observation, penalizing unresponsive nodes probabilistically. A node that suffers only a short outage might find that one copy of one of its objects has been lost, and can reestablish it from remaining replicas. However, a dishonest node will find it increasingly difficult to maintain its own objects in the system.

Samsara was designed as an extension to Pastiche [7], a cooperative backup system, but is applicable to other peer-to-peer storage systems. We have constructed a prototype of Samsara, and have evaluated it through benchmarking and simulation. The time required to exchange data is comparable to copying it via scp [27]. At low utilization, nodes have low exposure to risk of data loss, while as space becomes scarce, claims can be nearly eliminated. We explore the tension between space efficiency and reliability, and find that loss rates are modest—at 50% utilization, permanent and simultaneous failure of 16% of all nodes results in the loss of only 0.28% of all objects.

2. BACKGROUND AND CONTEXT

Pastiche [7] is a peer-to-peer, cooperative backup system. While greed—in the form of unchecked storage consumption—was identified as an important problem in Pastiche, it remained unsolved.

Pastiche nodes periodically take snapshots of their own file system state, and instantiate those snapshots on a set of remote replica nodes; as new snapshots are added, others can be retired. Each replica holds a complete copy of each retained snapshot. To minimize the storage overhead of this scheme, Pastiche nodes choose machines with state in common. For example, two Windows XP machines with the same set of applications already share most of their file state.

Pastiche identifies similar machines through the use of fingerprints. When node A needs a new replica, it hashes each of its distinct objects, and samples the set of hashes to produce a small fingerprint. This fingerprint is sent to a candidate node B to see what fraction of the fingerprint matches hash values of B's data. This fraction is a good predictor of the total amount of data in common between A and B.

Candidate nodes are selected with two mechanisms. The first is a systematic probe through a Pastry network [22], organized by network locality. Machines with common installations are likely to find suitable replica sites during this probe. Machines with less common installations join a second Pastry overlay, organized by

Pastiche	
Samsara	Pastry
OS, Disk	

This figure illustrates the relationship between Samsara, Pastiche and Pastry. Samsara sits between Pastiche and low-level storage. It does not use Pastry routing.

Figure 2: Peer-to-peer storage stack.

degree of data overlap. As a result of the Pastry join process, the machine is introduced to others with similar files. Once replica sites are identified, Pastiche no longer routes messages via Pastry, and uses IP instead.

Finally, each Pastiche node periodically checks its replica sites to ensure that backup data is retained. Only a subset of the data is checked during each period, providing a probabilistic guarantee. If any of these checks fail, the node instantiates a new replica and retires the old one.

Samsara was designed as a storage layer below Pastiche. All network messages in Samsara are sent directly over IP—a peer-to-peer message routing substrate is not needed. Figure 2 shows Samsara's relationship to Pastiche and Pastry.

3. DESIGN

Samsara's goal is to ensure that nodes consume no more resources than they contribute. Achieving Samsara's goal is straightforward if all storage relationships are symmetric. In a symmetric system, node A provides storage for node B if and only if B also provides storage for A. We call this *equal exchange*. B can periodically check its data to see if its data is held by A, and vice versa. Collectively, these pairwise checks ensure that each node consumes as much as it contributes.

Unfortunately, most peer-to-peer systems, including Pastiche, do not naturally conform to equal exchange. Samsara, therefore, provides storage claims—incompressible placeholders—to transform asymmetric storage relationships into symmetric ones. If node A stores data for node B, B must store an equal-sized claim for A. If B ever discards A's claims, A can discard B's data. To preserve equal exchange, B must be able to produce A's claim only if it is physically stored. A, however, cannot be forced to store local copies of its claim in order to verify that it is being honored. This would double A's storage burden.

Enforcing equal exchange using claims and simple querying is insufficient for a realistic system. First, no peer-to-peer system can assume that its participants are fully reliable, but the simple claims model punishes nodes severely for transient failure. Second, it doubles the storage requirements of the system, which could prevent nodes from creating replicas when storage utilization is high.

Peer-to-peer systems are composed of untrusted machines that can enter and leave the system at any time. The system ensures availability and durability by replicating objects at multiple nodes. To enforce equal exchange while also allowing transient failures, Samsara takes advantage of replication by punishing nodes probabilistically for failed queries. When a node fails a query, each of its replica sites independently drops data with some probability. If a node suffers only a short outage, it can restore any lost replicas from surviving copies. However, a cheating node that is never able to respond will suffer weaker data availability and durability as it fails more queries. Eventually, the cheating node will lose data.

Samsara reduces storage overhead when necessary by allowing claims to move freely throughout the system. When a node runs short of space, it can replace a claim it has given another node with one it is responsible for. This creates a dependency chain. For example, in Figure 1(c), B has replaced claim β_1 with claim γ_1, forming the chain $C \rightarrow B \rightarrow A$. This has implications for data reliability, because nodes who forward claims are still responsible for them. When a downstream node fails, upstream nodes are punished.

However, if a claim is forwarded back to its creator it forms a dependency cycle, and the claim can be removed from the system entirely. The resulting cycle tolerates single failures without cascading loss.

Finally, Samsara must be careful not to impose too much of a burden on nodes responding to queries. If queries are frequent and require extensive disk and network bandwidth to satisfy, the system will be unusable. Samsara makes an effort to minimize this burden by querying infrequently, allowing nodes to respond when convenient, and reducing response size to a single hash value.

3.1 Claim Construction

Samsara's claims transform each storage request into an equal exchange. To preserve equal exchange, nodes that are given claims must only be able to produce them if they are physically stored. Claim owners, however, must not be forced to store local copies of claims in order to verify that they are being honored. This would double the storage burden of the owner.

Before a node joins the system, it must initialize its storage space by logically filling it with storage claims. Later, when data is placed in the storage space, the node can return any claims that were overwritten.

Computing a claim requires three values—a secret passphrase P, a private, symmetric key K, and a location in the storage space. To initialize the space, nodes first logically fill the storage space with hash values. In the first 20 bytes, nodes compute the SHA1 hash of the concatenation of the passphrase P and the number 0, denoted $h_0 = SHA1(P, 0)$. For each 20 byte chunk that follows, the hash $h_i = SHA1(P, i)$ is stored in the ith chunk.

Claims are fixed-sized blocks formed from consecutive hash values. For example, suppose that claims are 512 bytes long. The initial claim, denoted C_0, would be computed by concatenating the first 25 hashes with the first 12 bytes of the 26th hash, and then encrypting it using the symmetric key K. The second claim, C_1, would be computed by encrypting the concatenation of the next 25 hashes and the first 12 bytes of the hash after those, and so forth. Thus, the $i$the claim, C_i, is :

$$C_i = \{h_j, h_{j+1}, \ldots, h_{j+24}, h_{j+25}[0], \ldots, h_{j+25}[11]\}_K$$

where $j = i \times 26$.

Of course, claims do not have to be computed during initialization, and never need be stored on the originating host. They can be computed on the fly as needed.

3.2 Querying Nodes

Querying allows nodes to monitor their remote storage. Because querying is only intended to demonstrate that data is stored rather than to examine its contents, queries can be made relatively infrequently. A node could reasonably query its replica sites every few hours or even once a day.

Similarly, queries need not be answered immediately. This is important because nodes might want to delay responding to a query when they are busy. There are many ways to minimize the burden of answering a query, including resource containers [3], progress-based mechanisms [11], and free disk bandwidth [20]. The load of responding to a query is quantified in Section 5.1.
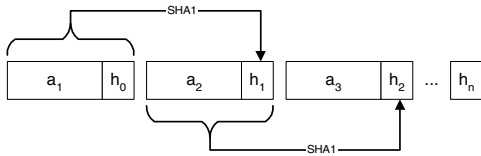
This figure demonstrates how a node can respond to a query. The querying node supplies an initial value h_0, which is appended to the first data object, a_1, and hashed to produce h_1. This is then appended to the second object, a_2, and hashed, and so on. The storing node only needs to return the final hash h_n.

Figure 3: Query response construction.

The network bandwidth required to satisfy a query can be reduced to a single SHA1 hash as well. There is no need to return the entire data object to prove that it is still stored. When querying, nodes can send a unique value, h_0, along with the list of n objects they wish to verify. Responding nodes append h_0 to the first object in the list and compute the SHA1 hash of this concatenation. This hash is called h_1 and is appended to the second object to compute h_2 and so on. The responding node only needs to return h_n to prove that it is storing all the objects it is responsible for. This process is shown in Figure 3.

Nodes must maintain three lists to perform queries. The *location list* maps object identifiers to locations in the storage space, the *storage list* maps node ids to ids of objects in the storage space, and the *replica list* maps node ids to replicated object ids. Nodes do not have to maintain a list of sites storing their claims. This information can be recreated using the storage list and location list. To query for a set of claims, a node walks the list of objects stored locally for a given host and recomputes the claims exchanged for those objects using their location in the storage space.

Nodes must be careful that queries cannot be answered by someone other than the intended node. This is easy to do for claim queries, as claims are not replicated. For data, replicas must be salted to make them unique to their intended storer.

3.3 Transient Failure

Nodes in peer to peer networks suffer from frequent transient failures [5]. Distinguishing between a node that is trying to cheat and one that experiences a transient failure is both difficult and important. Nodes should not lose all of their remote data because of a temporary loss of connectivity or failure. This is especially true for a backup service like Pastiche. When a node's disk dies, it is unable to respond to queries. If the node's remote data is then discarded because it fails a query, restore will be impossible!

One potential solution requires nodes to provide free storage for a *grace period* up front as part of the cost of setting up partnerships. At the onset of a partnership, a node will be unable to retrieve any stored data. Unfortunately, this restriction is unacceptable for many applications.

Instead of restricting access to stored data for a grace period, nodes can grant a grace period for responding to a query. If this grace period is longer than a conservative estimate to recover a failed node—on the order of many days to weeks—then honest but unfortunate peers will not be penalized. This scheme tolerates transient failure while allowing immediate access to stored data.

Unfortunately, it also leads to a straightforward attack [18]. A node could choose replicas for grace periods at a time, never storing anything in return, and select new peers once the grace period expires. This is easy to do in systems like Pastiche, where nodes have full discretion in selecting replica sites. It is also possible to mount this attack in a system that is more constrained in selecting

replica sites. For example, nodes could encrypt their objects under a rotating key scheduled every grace period. The resulting objects have the same semantic content as their source, but will have different object identifiers, and will thus be stored on different replica sets.

The problem with the grace period approach is that it treats the grace period as an all-or-nothing deadline. Samsara retains the notion of a grace period, but it is a gradated one.

Recall that peer-to-peer storage systems rely on replication to provide high availability in the face of failure. Samsara uses replication to punish failed queries probabilistically. The number of replicas owned by a node should decay as it fails queries. A data object is lost if all replicas of that object disappear. If a node fails a single query at all replicas, only one replica should disappear, but the node should not lose its data. A node should only lose all of its data if it fails queries for an entire grace period.

Samsara accomplishes this through independent, probabilistic discards of an unresponsive node's object. Each node discards with some probability p_i, where p_i is chosen based on the number of failed queries and the normative replication factor, denoted r, of the system. Probabilistic discard does not prevent nodes from mounting an isolated grace period attack, but it does prevent long-term abuse. Nodes who perpetually abuse the grace period can expect to lose data.

Assume that we want an unresponsive node to lose one replica for each consecutively failed query. For the ith consecutive failed query, each replica site computes

$$p_i = \frac{1}{r - i + 1}$$

If the unresponsive node has n replicas, the expected number of discarded replicas after failing the first query of each replica site is $D_1 = \frac{n}{r}$. More generally, $D_i = \frac{n - D_{i-1}}{r - (i-1)}$ discards are expected after failing the ith consecutive query of each site. Thus, $D_r = n - D_{r-1}$, which means that after r consecutive failed queries at each site, all n replicas should be discarded.

There are two complimentary strategies for defeating probabilistic discard that immediately come to mind. First, as in the original grace period attack, a cheating node could create a brand new set of replicas frequently. In addition to this, a node could create many replicas to increase the expected number of consecutively failed queries before losing an object. A cheating node would have to create new replicas fast enough to ensure that there is at least one copy of each object at all times. As we will see, the bandwidth required to create enough replicas fast enough is prohibitive for most nodes.

Our first computation of p_i was over-simplified. If p_1 were actually $\frac{1}{r}$, a node with n replicas and one object can expect all n replica sites to simultaneously discard its object after missing one query, with probability $\frac{1}{r^n}$. This means that for $r = n = 5$, a node can expect to lose an object after missing a single query at each replica site on 3,125 separate occasions. This is far too severe, particularly since nodes are likely to have many more than one object, and replica sites roll the dice for every object stored when a query is missed.

A typical Pastiche node will have about 2,000,000 objects, equal to about 32GB of state. Thus, when $r = 5$ and

$$p_1 = \frac{1}{r}$$

a Pastiche node with five replicas can expect to lose 640 objects the first time it misses a query at each replica site. This is clearly inadequate, even if nodes are given a day to respond to each query.
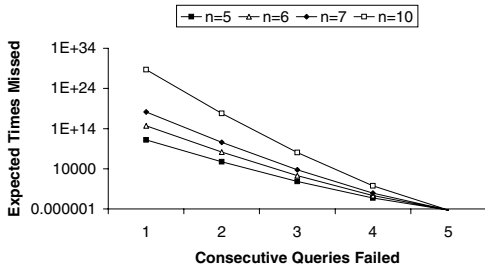
This figure graphs query generation versus the expected number of times a node can miss that many consecutive queries before losing an object. The normative replication factor for the system is assumed to be five ($r = 5$), but the adversary populates n replicas. Each node owns 2,000,000 objects. Each series represents the number of replicas actually created.

Figure 4: Expected missed queries before object loss.

Instead, failures should be exponentially more harmful as consecutive queries are missed to protect honest nodes while still punishing cheaters.

One way to refine p_i is as follows :

$$p_i = \left(\frac{1}{(r-i+1)}\right)^{(r-i+1)}$$

For $r = 5$, a node with 2,000,000 objects can expect to lose an object after missing a query at each replica site 1.49×10^{11} times. If queries are made on a daily basis, this would require 4.08×10^8 years of always missing a query. The expected number of consecutive queries that can be missed before losing an object is graphed for various replication factors in Figure 4. As the figure shows, cheating nodes will be punished eventually, even when the replication factor is doubled.

With $n = r = 5$, 2,000,000 objects, and our modified p_i, a node can expect to miss three consecutive queries at all replica sites only seven times before losing an object. Adding an additional replica raises this number to 194, which is reasonable. Even a behaving node will occasionally miss three consecutive queries, but it will happen infrequently because most nodes usually only experience outages of less than three days [5].

These curves descend quickly, however. The expected number of lost objects after missing four consecutive query jumps to 488. Because of this, for the first three days, queries should be scheduled everyday, while the fourth query should be scheduled a week later. Cheating nodes can still expect to eventually lose objects after just three days because of the number of times they will miss three consecutive queries.

Still, as alluded to earlier, a cheating node might be able to defeat probabilistic discard by creating many replicas and then replacing them before it expects to lose an object. A node with seven replicas can expect to lose an object after missing three consecutive queries 5,230 times. This is fairly risky. However, a node with ten replicas can expect to miss three consecutive queries over 100 million times before losing an object. For this node, missing three consecutive queries is probably safe, even if it expects to lose two objects if it fails four consecutive queries. Fortunately, the bandwidth required to create seven new replicas every three days or ten new replicas every ten days is prohibitive for most nodes.

Figure 5 shows the continuous outbound bandwidth required for various combinations of replication factors and days to create those replicas. Assume that a cheating node has 2,000,000 objects and 32GB of state. For a node with five replicas, it will need to create approximately five replacement replicas within three days to avoid
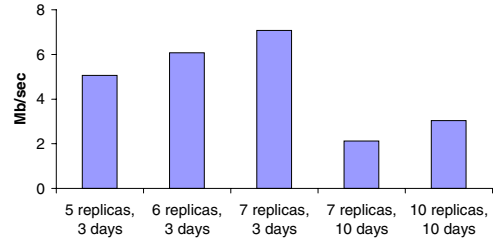
This figure shows the continuous outbound bandwidth required to create five, six, seven, or ten replicas every three or ten days. Nodes are assumed to have 32GB of state and a normative replication factor of five.

Figure 5: Required bandwidth to create replicas.

losing an object. To create five new replicas within three days a node to ship 160GB of data, for a continuous outbound bandwidth of approximately 5Mb/s. If the node tries to maintain six replicas, the bandwidth requirement is about 6Mb/s. Even a node with ten replicas, who is unlikely to ever lose any objects after missing three consecutive queries, needs over 2Mb/s to recreate ten replicas over ten days.

Bandwidth between arbitrary points on the Internet is unlikely to approach these levels anytime soon [19]. Furthermore, network bandwidth is a far scarcer, and thus more valuable, resource than storage. Paying for remote storage with bandwidth instead of storage isn't cost effective.

However, while bandwidth constraints will likely keep nodes with data sets of 32GB or larger from cheating, this might not be enough to prevent nodes with only a few objects from cheating. The less data a node needs to replicate, the less bandwidth it needs to create new replicas every three or ten days. While this is true, nodes with smaller data sets are, by definition, unlikely to consume large amounts of storage.

Nonetheless, if nodes with smaller data sets become a problem, one could imagine further refining p_i. One strategy is to identify which nodes are capable of cheating. By using the state size and observed bandwidth, nodes could approximate the maximum number of replicas that another node is capable of creating before it expects to lose an object. The discard rate could then be set higher for nodes with enough resources to cheat. Of course, simply being capable of cheating does not mean that a node will. We plan to explore this technique more in the future.

3.4 Reducing Claim Overhead

When node A stores data for node B and B stores for A, we say that B is *downstream* of A; claims always move downstream. The exchange between A and B represents an exchange of *storage rights*. Each node is entitled to store anything it chooses on the remote disk blocks of the other. A can replace its object with anything and B can replace its claims with anything. Replacement does not affect equal exchange since all nodes still consume and contribute equally.

To authenticate replacement, the two nodes negotiate a Diffie-Hellman key [9] during initial exchange. This is an unauthenticated exchange because we do not assume that nodes have certified identities. It is therefore subject to man-in-the-middle attacks. However, if the initial exchange is not intercepted, future replacement requests can be checked for authenticity. Samsara decreases the likelihood of interception by routing negotiation messages via IP rather than the overlay.

In the simple case where nodes never replace the claims they create, the storage overhead of Samsara is equal to the data stored in

(a) A simple data storage network

(b) Forwarded claim

(c) Claim cycle
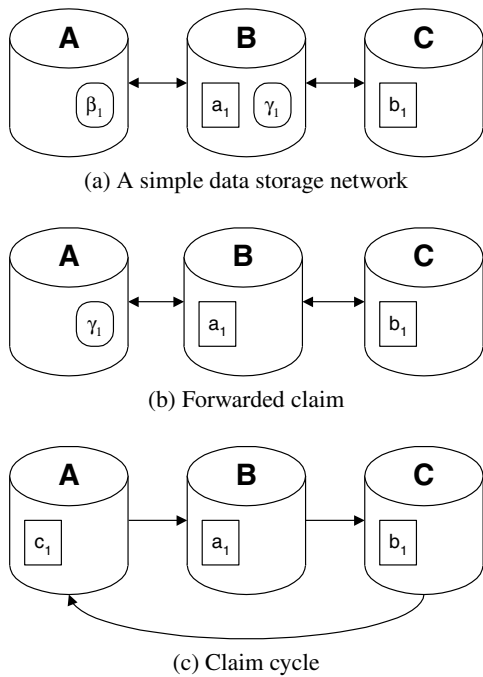
This figure illustrates how a claim is forwarded in a simple three node storage network. Arrows represent query lines. Objects are boxes and claims have rounded corners. Node C is storing object b_1 for node B. B is storing claim γ_1 for C in exchange. B is also storing object a_1 for node A, who is in turn storing claim β_1 for B. Because B wants to free up space so that it can create a second replica of b_1, it replaces β_1 on A with γ_1 and forwards any queries or storage requests for γ_1 from C to A. Say that C wants to create a replica of object c_1 on A. This allows A to replace whatever new claim it would have returned to C in exchange with claim γ_1, which removes γ_1.

Figure 6: Forwarding claims.

the peer-to-peer storage system, doubling global storage requirements. When storage becomes scarce, however, nodes can use the storage rights they have gained by creating claims. They do this by replacing the remotely stored claims they have created with locally stored claims. Moving claims further downstream is called *claims forwarding*; forwarding frees local storage.

Consider the network shown in Figure 6(a). Node A stores its object a_1 on node B, which in turn stores claim β_1 on A. B also stores object b_1 on node C, which in turn stores claim γ_1 on B. B only has space for two blocks—both of which are currently occupied—but wants to create a second replica of b_1. Creating a second replica is possible only if B has enough space to store another node's claims. Thus, B needs to free a block of storage.

B can free a block by using the storage rights it received for storing a_1, so B replaces β_1 with γ_1. B no longer needs to query for β_1. Whenever C queries B for γ_1, B will simply forward the query to A. This network is shown in Figure 6(b).

Allowing nodes to transfer claims does not enable the Sybil attack [10]. Suppose a dishonest node creates an alias for itself. When it receives a claim, it can "forward" the claim to its alias, but the node gains no advantage. It must still produce the claim when asked, and must therefore still store it.

B was able to eliminate one claim, but another claim, γ_1, still exists. Now say that C wants to create a replica of its object c_1 on A. A first stores c_1 for C and returns claim γ_1 to C, rather than creating a new claim. When γ_1 is forwarded back to its owner, C,

a cycle is created, and γ_1 can be removed. This network is shown in Figure 6(c).

Even when a cycle is created, each node's storage rights remain intact. C still has the right to a block on B, B still has the right to a block on A and A has the right to a block on C. In a cycle, every node can forward a replacement request to a downstream node until the request comes back to the node that originally made it.

The cycle example, as presented, uses the simple optimization of telling nodes where their claims are physically stored. Without the optimization, C would initially receive a brand new claim, α_2, from A. C would then use its storage rights on B to forward α_2. B would in turn forward α_2 to A, who does not need to store the claim because it can compute it. The result is the same—all claims are removed. When C knows that A is storing γ_1, all of this forwarding can be eliminated. C can simply tell A to return γ_1 instead of creating new claim α_2. This optimization has the additional benefit of eliminating query forwarding since nodes can query claim storers directly.

3.5 Forwarding and Reliability

When a node forwards a claim downstream, it remains responsible for it. Without authenticated identities, nodes cannot prove to the claim owner that the claim was forwarded. To see why, consider a dishonest node that can create aliases for itself. Every claim received by this dishonest node could be "forwarded" to one of its fictitious identities. If other nodes believed these assertions, the dishonest node would never be held accountable for discarding the claims it should have been storing.

Samsara assumes that nodes respect the storage rights they have promised other nodes when their own data depends on it. In the case where claims are never forwarded, dealing with failure is easy. Whichever node's query failed must be storing data owned by the failed node. The querying node simply discards the failed node's data.

When claims are forwarded, however, the node that detects the failure might not be storing any data owned by the failed node. Consider the five node dependency chain in Figure 7(a). In this network, a dependency chain exists via claim ϵ_1. Node A is downstream of node B, B is downstream of node C, C is downstream of node D, and D is downstream of node E.

Suppose that C fails B's query. B is not storing c_1, so it cannot discard C's data. However, B still has storage rights on A, so it uses those rights to replace ϵ_1 with its own claim, β_1. A must replace ϵ_1 with β_1 to protect its object, a_1.

Meanwhile, E holds D responsible for ϵ_1. When E queries D for ϵ_1, the query fails. E then discards D's object, d_1. In response, D attempts to use its storage rights at C to replace ϵ_1 with a claim of its own. When this request fails, D discards C's object, c_1. The resulting network is shown in Figure 7(b). All data upstream of the failure is lost, but downstream data is preserved.

In our example, even though D did not fail, it lost d_1 as a result of C's failure. Forwarding a claim makes the data exchanged for that claim only as reliable as the weakest downstream node. The longer the dependency chain, the less reliable each upstream node's data becomes. For this reason, Samsara nodes only forward claims when absolutely necessary—when a node's storage is near capacity. This creates a strong incentive for nodes to add storage. The more storage nodes dedicate, the less likely they are to forward claims, and the more reliable their remote state will be. Even though dependency chains weaken reliability, as we show in Section 5.4, object loss is still rare, unless both the permanent failure rate and storage utilization are very high. Under normal conditions, we expect most failures to be transient and storage utilization to be moderate.

125

(a) before failure

(b) after failure
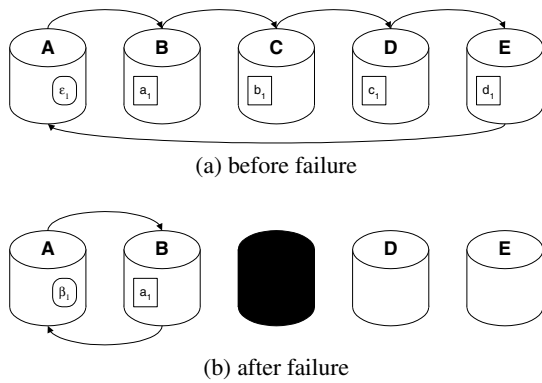
This figure illustrates a five-node dependency chain of claims, both before and after the middle node fails. Arrows represent query lines. Objects are boxes and claims have rounded corners. Upstream nodes lose all data, while downstream nodes retain it.

Figure 7: Failure in dependency chains.

(a) before failure

(b) after failure

This figure illustrates a five-node dependency cycle of claims, both before and after the middle node fails. Arrows represent query lines. Objects are boxes and claims have rounded corners. Upstream and downstream nodes retain all data.

Figure 8: Failure in dependency cycles.

Message	Arguments	Return Value
store	$pathname, objectid, location, sync$	$error$
retrieve	$objectid, location, sync$	$pathname$
query	$location$	$error$
callback	$pathname, event$	id

(a) `samsarad` messages.

Event	Description
store_req	a remote node requests storage
retrieve_req	a remote node requests data
query_res	a query succeeds or fails

(b) Callback events.

This table shows the messages that can be passed to the `samsarad` daemon by higher level peer-to-peer storage systems, like Pastiche. Each message is passed to `samsarad` via Unix domain sockets. The `callback` message allows other processes to register a socket to be notified when an event occurs. The events that can be registered for are listed and described in the bottom table.

Figure 9: Samara interface.

Surprisingly, while dependency chains weaken reliability, cycles actually restore it. This is because each node in a cycle has storage rights on another node in the cycle. In a dependency chain, the node farthest downstream does not have any storage rights. When the farthest node does have storage rights, data can "wrap around" any failure. Consider the five node dependency cycle in Figure 8(a). In this network, a cycle exists where node A is downstream of node B, B is downstream of node C, C is downstream of node D, D is downstream of node E, and E is downstream of A. Suppose, as before, that C fails B's query. B again uses its storage rights to store claim β_1 on A.
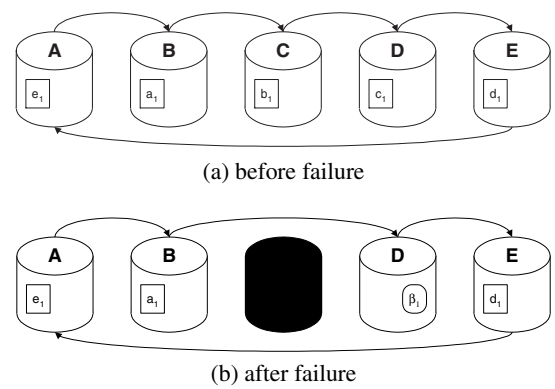
Before, A replaced ϵ_1 with β_1. Now, however, A has storage rights on E, because A is storing E's object e_1. A can use those rights to forward β_1 to E. E has storage rights on D and thus forwards β_1 to D. D has storage rights on C and attempts to store β_1 on C. This fails, so D discards C's object, c_1. However, because D wants to protect its object, d_1, on E, it stores β_1. The resulting network is shown in Figure 8(b). This time all data upstream and downstream of the failure remains intact.

This puts Samsara in a strange predicament. Forwarding weakens data reliability as long as the claim remains active in the system, but if the claim can be retired to create a cycle, reliability is restored. It is thus in nodes' interest to create cycles, but not chains. It is unclear how Samsara can induce cycles without constraining replica choice or creating chains, but it presents an interesting direction for future work.

3.6 Limitations

There are a number of issues facing peer-to-peer storage that Samsara does not solve. First, our target is greedy users, not arbitrarily malicious ones, because we believe that greedy users are far more common. The benefits of being greedy are tangible while the benefits of being malicious are not. Thus, Samsara does not stop nodes who promise to store data and then immediately discard it. Similarly, Samsara cannot prevent a Pastiche node from refusing service once it detects that a restore is in progress.

A more important limitation of Samsara is that while it solves the problem of unchecked consumption, it does not necessarily solve the problem of contribution. Nodes are forced to provide storage, in the form of claims, equal to the storage they consume. However,

Samsara cannot force nodes to store data for others. If all nodes refuse to store anything other than the claims they receive, data will never be stored. Nonetheless, storing others' data can be a way to temporarily access remote storage when it is difficult to find. If a node has trouble finding replica sites, it can accept storage requests from other nodes and replace its claims with actual data until it finds satisfactory sites.

Finally, the technique of creating and verifying resource placeholders appears to only be applicable to non-renewable resources like storage and memory. It is difficult to imagine how one would create placeholders for renewable resources like bandwidth and processor cycles. Even if a placeholder could be created, monitoring its maintenance would consume the resource itself.

4. IMPLEMENTATION

We have built a prototype of Samsara, written in C, called `samsarad`. The daemon is composed of three layers—a messaging layer, a replica manager, and a storage layer.

The messaging layer is responsible for sending and receiving all network and local messages. Local messages are passed via Unix domain sockets. This is how Pastiche communicates with Samsara.

There are four messages that can be sent to `samsarad`—`store`, `retrieve`, `query`, and `callback`. The arguments and return values for these messages are summarized in Figure 9. All network communication uses the RPC2 package [25].

Once the messaging layer receives a message, it forwards the request to the replica manager. The replica manager is responsible for authentication and maintaining replica locations. We used the `openssl0.9.7` library for all authentication including a 2048-bit Diffie-Hellman key exchange with SHA1 hashes.

The storage layer is responsible for knowing who owns any stored data and where that data lies; it also handles claims generation, using a claim size of 4096 bytes.

For expediency, the storage space is a single, flat file, stored in the underlying `ext2` file system. The storage layer maintains two persistent free lists to help place new data—one for unused storage and one for storage occupied by claims. These lists are currently only linearly searched, single-level lists. We do not currently deal with fragmentation or make any attempt to optimize data placement. We simply choose the first free location that is large enough to hold whatever we are trying to store.

All object locations are stored in a database that maps object identifiers to a size and offset within the storage space. Additionally, all stores are whole-object grained. In the future, we plan to break objects into claims-sized blocks to better facilitate claims transfer.

5. EVALUATION

In evaluating Samsara, we set out to answer the following questions:

- What are the data transfer and query performance of the prototype?
- How much processor and disk overhead does querying induce?
- How does storage utilization affect the need to forward?
- How often do dependency cycles arise naturally?
- How long do dependency chains get as utilization increases?
- How do dependency chains affect reliability?

5.1 Prototype Micro-benchmarks

We were interested in measuring two aspects of our prototype, `samsarad`. We wanted to know what its data transfer and querying performance were and we wanted to characterize the load of responding to and verifying queries. To explore both questions, we used two representative data loads. The first load, referred to as the "tree" load, consisted of 1676 files totaling 13MB. The other load, referred to as the "archive" load, consisted of a single 13MB file. We chose these two loads to examine the effect of transferring and querying many objects versus just one. The data used for both was taken from the `openssl-0.9.7a` source tree. The tree load was all regular files in this source tree, while the archive load was all regular files concatenated into a single file.

All experiments were run on machines with a 550 MHz Pentium III Xeon processor, 256MB of memory, and a 10k RPM SCSI Ultra wide disk. The disk has an average 4.7 ms seek time, 3.0 ms rotational latency, and 41 MB/s peak throughput. Testbed machines are connected by a switched, 100Mb/s Ethernet fabric.

We first measured data transfer performance. How does `samsarad` perform relative to the commonly-used secure copy utility `scp`? In this experiment, `samsarad` first copied data to a replica site. It then retrieved and stored the claims exchanged for that data. We ran ten trials per experiment with both `samsarad` and `scp`, being careful to flush the system's buffer cache between trials. The performance results are in Figure 10.
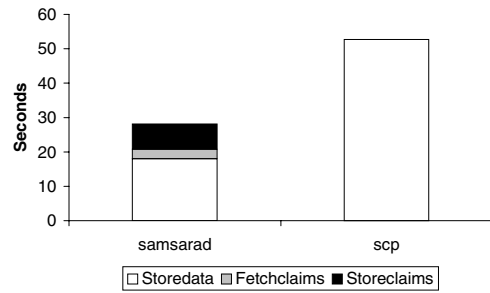

(a) Time to complete tree benchmark.
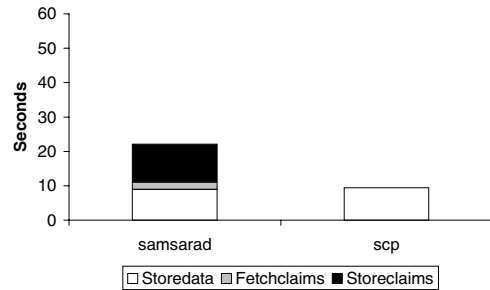
(b) Time to complete archive benchmark.

This figure shows the time to transfer both the tree and archive loads for `samsarad` and the copying utility `scp`. The Samsara prototype requires three phases to complete the transfer—storing the data at a replica site, receiving claims in return, and then storing those claims.

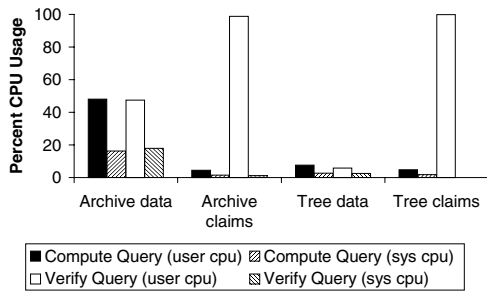**Figure 10: Data transfer performance.**
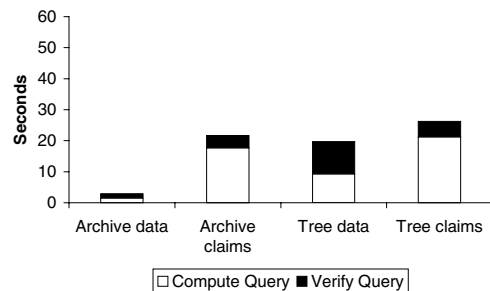
This figure shows the time to compute and verify a query under both the tree and archive loads. Times to compute and verify queries for both the data and the claims exchanged are shown.
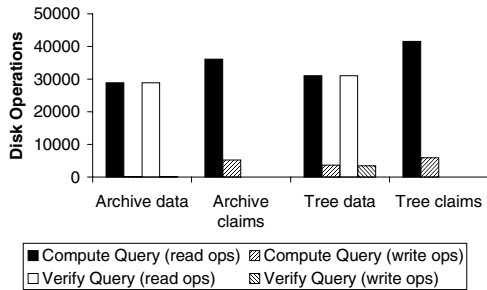
Figure 11: Query performance.

`samsarad` outperformed `scp` for both loads during the store data phase, with `scp` performing much worse under the tree load. Storing claims, however, is quite slow, taking nearly 10 seconds to complete. The reason for this is that `samsarad` needs to store each claim, and then update the claim's entry in the storage location database. While slow, this performance is still reasonable since most peer-to-peer storage systems ship data asynchronously.

We evaluated query performance and overhead next. We queried the data and claims associated with the tree and archive loads. This involves two phases—the query response computation and verification. The results are shown in Figure 11. These measurements were taken after the data had been copied and the buffer caches at both machines had been flushed.

For both data queries, the time to compute the response was the same as the time to verify it. This is because both the querying

(a) CPU load during querying.

Legend: ■ Compute Query (user cpu) ▨ Compute Query (sys cpu) □ Verify Query (user cpu) ▨ Verify Query (sys cpu)

(b) Disk operations during querying.

Legend: ■ Compute Query (read ops) ▨ Compute Query (write ops) □ Verify Query (read ops) ▨ Verify Query (write ops)

This figure shows processor and disk load during query computation and verification.

Figure 12: Query processor and disk load.

Collection-based replication					
	50%	57%	67%	80%	100%
replicas	20000	20000	20000	20000	19997
claims	13626	11488	8619	4740	3
cycles	86	129	173	180	152
forwards	6326	8528	11942	18913	131360
DHT replication					
	50%	57%	67%	80%	100%
replicas	20000	20000	20000	20000	19998
claims	15207	12753	9168	9168	2
cycles	5	7	4	6	37
forwards	4788	7269	11332	18050	119833

This table shows the storage overhead in two simulated Samsara networks of 5000 nodes, each needing four replicas. Each column represents the percent of global storage needed to store all replicas.

Figure 13: Simulated space overhead.

and storing node performed the same tasks. Each needed to read the data from disk to compute the response. It took much more time to compute and verify the tree data, because it required more disk reads. For claims queries, the time to compute a response took much longer than the time to verify. This is because verifying a claims query is done entirely in memory, which is much faster than reading from disk.

The processor and disk loads of computing and verifying queries are shown in Figure 12. The processor and disk loads of the querying and responding nodes are the same for data, but the overhead of responding to a claims query is very different from verifying it. The responding node uses many more disk operations to read the claims from disk, while the querying node requires more processor time to recompute the claims.

5.2 Space Overhead

In the worst case, when nodes never transfer claims, Samsara doubles the global storage burden. This could be problematic when storage utilization is high. To explore how effectively Samsara can reduce its overhead under high utilization, we built a Samsara simulator on top of SimPastry. We populated our simulator with 5,000 nodes, and set the replication factor to four. Each node was assumed to have one object that was equal in size to a claim.

We were also interested in the relationship between replication strategy and overhead. Objects can be replicated as individual objects in a distributed hash table (DHT), or can be replicated as collections on nearby nodes. The former strategy is employed by CFS [8] and PAST [23], while latter strategy is similar to that used by cooperative backup systems [7, 18]. For both replication strategies, we varied system-wide space utilization from 50% to 100%. In each experiment objects were forwarded to other nearby nodes, as determined by replication strategy, if their "natural" replication sites were full. Each experiment continued until all objects were placed or storage was exhausted.

The results are in Figure 13. Nodes were almost always able to replicate their objects, even at 100% storage utilization. In the 100% utilization case, both collection-based and DHT replication reduced claim overhead to less than 0.02% of global storage.

Even though collection-based and DHT replication pursue very different strategies, claim overhead can be effectively reduced in both. The biggest difference between collection-based and DHT replication is that collection-based generated more cycles than DHT. The reason for this is that collection-based nodes are more likely to choose one another. The collection-based strategy targets replica sites that are close in the network, which is generally symmetric. This leads to clustering of nodes that are close to each other, and makes cycles more likely. In DHT replication, nodes replicate along a random range of the nodeid space. The randomness makes it highly unlikely that nodes within a range of the nodeid space will choose to replicate their own objects on reciprocal peers.

Still, for both replication strategies, we observed very few cycles. Because chains rarely become cycles, forwarding claims will almost always weaken data reliability.

5.3 Chain Length Analysis

Samsara can remove enough claims to ensure that all nodes are able to create replicas, no matter the storage availability. However, there is a tension between the length of the transfer chains needed to clear storage and the reliability of the replicas created along that chain. How long did these chains actually grow? Figure 14 shows the distribution of chain lengths under both collection-based and DHT replication. These results are summarized in tabular form in Figure 15.

Chain lengths grow beyond eight only when approaching 100% utilization. In the 100% utilization case, 69% of chain lengths are less than four and 79% are less than eight. Interestingly, chains can become extremely long when utilization requirements are 100%. In one experiment, for example, we witnessed a chain that was 3,822 nodes long out of 5,000.

We must know two things to explain why this happens—how and under what circumstances chains become longer. When a claim is forwarded, its chain lengthens by the size of the chain it replaces. The sum of all chain lengths increases when claims are introduced, remains unchanged when claims are forwarded, and declines through cycles.

Few new claims are introduced beyond a certain point, cycles are rare, and storage is cleared almost exclusively by forwarding claims. What this means is that the sum of all chain lengths among active claims remains relatively constant. Because this sum re-
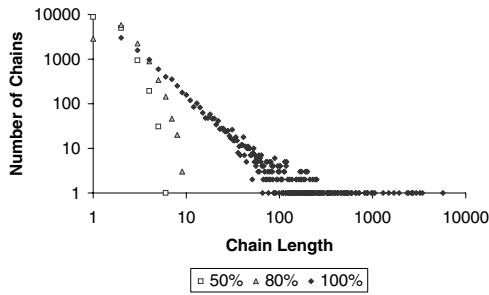
(a) Distribution of chain lengths for collection-based replication.
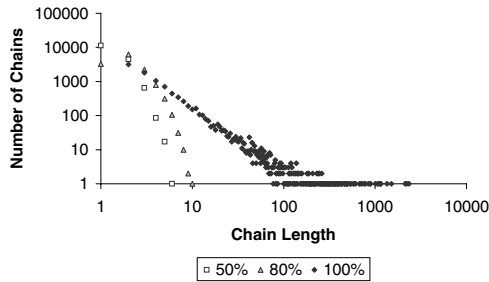
(b) Distribution of chain lengths for DHT replication.

This figure shows the distribution of chain lengths for networks using collection-based and DHT replication. Each series represents a percent storage utilization required to create all replicas.

Figure 14: Chain length distributions for claims transfer.

| Collection-based replication			
	80%	50%	100%
total claims	14971	10671	9098
% with length ≤4	1	.88	.68
% with length ≤8	1	.97	.79
DHT replication			
	80%	50%	100%
total claims	16529	11126	9775
% with length ≤4	1	.81	.62
% with length ≤8	1	.98	.80

Figure 15: Chain length distributions.

mains stable, the fewer active claims there are, the longer their chains necessarily become. In addition, when storage utilization is 100%, there are between four and five times more forward operations then when utilization is 80%. Forwarding reduces the number of claims without changing the sum of all lengths, which leads to longer chains.

5.4 Reliability

Chain length is important because it has implications for reliability. The reliability of data stored along a transfer chain depends on downstream nodes not failing. Because of this, it is important to explore the relationship between storage utilization, chain length, and data reliability.

To evaluate reliability in the face of permanent node failure, we constructed a 5000-node network using DHT replication. Once constructed, we assumed that nodes *permanently* and *simultaneously* failed with some probability. This eliminated the failed nodes' objects along with any objects stored on other nodes in exchange. If any lost objects were forwarded claims, all upstream data was lost, as described in Section 3.5.
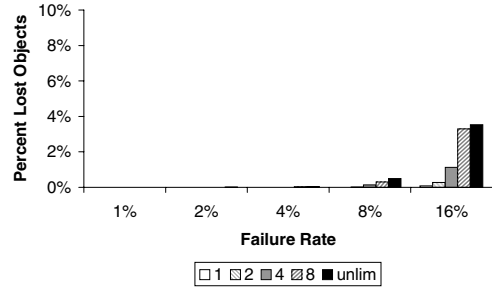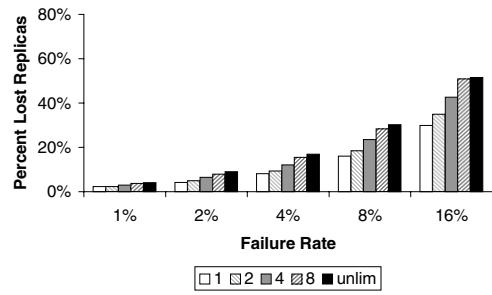
This figure shows the percent of lost replicas and objects when chain lengths are capped using aggressive claims transfer. An object is lost only when all of its replicas are.

**Figure 16: Reliability of data with capped chain lengths.**

Our first set of experiments isolated the effect of chain length on reliability. To do this, we gave nodes unlimited storage capacity and transferred claims aggressively, up to a limit on the number times claims could be forwarded. The results of these experiments are in Figure 16. It shows both the fraction of lost replicas and the fraction of lost objects; an object is lost only when all of its replicas are.

We are interested in the percent of lost replicas beyond those lost in the base case, where transfers are capped at one. The number of lost replicas for the base case reflects both the objects stored *at* the failed node and the objects stored *in exchange* for the claims stored at the failed node.

There is a significant drop in reliability between chains of length four and eight. For a permanent failure rate of 8%, when chain lengths are limited to four, only 0.13% of objects are lost, whereas when chain lengths are capped at eight, 0.30% of objects are lost. When the permanent failure rate is 16%, 1.1% of objects are lost when chain lengths are four or less, while 3.3% are lost when chain lengths are eight or less. Because forwarding claims only when necessary keeps almost all chain lengths below four, it will be reliable even with widespread failure.

Our second set of experiments focused on the effect of storage utilization on reliability. When storage utilization is low, nodes hold onto all claims they receive, rather than endanger their data by transferring them. When storage utilization is high, nodes are forced to transfer them. To understand this interaction, we examined a range of failure rates for storage utilizations of 50% and 80% and transferred claims both conservatively and aggressively. The conservative trials represent expected reliability, while the aggressive trials represent worst-case reliability. The results are in Figure 17.

These figures show that nodes should always forward claims conservatively. Transferring claims aggressively is no more effective at clearing space for replicas at high utilizations, but increases the probability that an object will be lost several-fold. The fig-
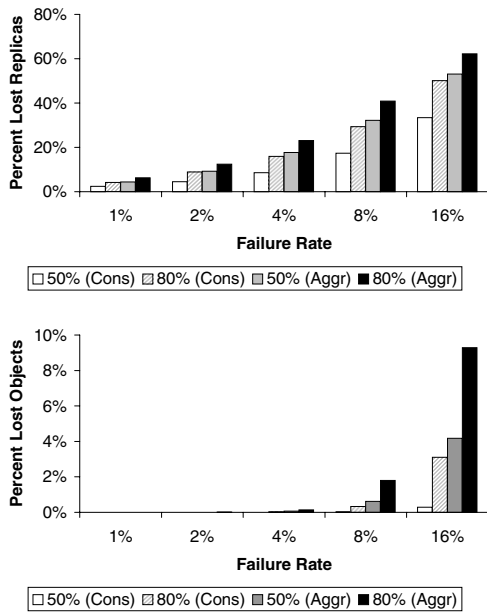
This figure shows the percent of lost replicas and objects for utilizations of 50% and 80%, paired with both aggressive and conservative claims transfer schemes.

Figure 17: Reliability of data under various storage utilizations.

ures also show that excess storage can be adaptively occupied to increase reliability, or cleared to create space for more replicas. When utilization is at 50%, transferring claims conservatively and using that excess space to store claims keeps object loss rates low for all failure rates. Even when 16% of all nodes permanently fail, only 0.28% of objects are lost. The effect of utilization on reliability gives individual users an incentive to dedicate more local storage to the system.

It should be noted that our simulations are pessimistic. We have assumed that all failures are permanent, because we provide mechanisms for preserving data in the face of transient ones. Studies have shown that while transient failures are common, permanent ones happen far less frequently [5]. It is unlikely that even 1% of a large population of nodes would all permanently fail in such a short period of time that the affected nodes are unable to find replacement replica sites.

6. RELATED WORK

The problem of fairness in peer-to-peer systems is an instance of the *tragedy of the commons* [15]. In short, the benefit of using the system accrues individually, but the costs are shared. A rational participant concludes that it is in his best interest to consume as much as possible without contributing, allowing others to pay his way. Any such system is not sustainable in the long term.

This is not merely of academic interest. A study of Gnutella [1] found that two thirds of the participants benefited from the system without contributing to it. Furthermore, the most generous 1% of hosts serviced nearly half of all requests. A later study of Gnutella and Napster [24] confirms this general trend, though with a less dramatic fraction of non-cooperative nodes. This study also found that almost one third of all Napster users under-reported their available bandwidth to avoid being selected as a site from which others would download files.

Some mechanism is required to enforce fairness amongst peers—

that they contribute in proportion to what they consume. To the best of our knowledge, Samsara is the only system that enforces fairness in peer-to-peer storage without requiring trusted third parties, symmetric storage relationships, monetary payment, or certified identities.

Several peer-to-peer storage systems have considered the problem of enforcing storage quotas in some way. Two of the first to do so were CFS [8] and PAST [23].

In CFS, each storage node limits any individual peer to a fixed fraction of its space. These quotas are independent of the peer's space contribution. CFS uses IP addresses to identify peers, and requires peers to respond to a nonce message to confirm a request for storage, preventing simple forgery. This does not defend against malicious parties who have the authority to assign multiple IP addresses within a block, and may fail in the presence of network renumbering [12].

PAST provides quota enforcement that relies on a smartcard at each peer [23]. The smartcard is issued by a trusted third party, and contains a certified copy of the node's public key along with a storage quota. The quota could be determined based on the amount of storage the peer contributes, as verified by the certification authority. The smartcard is involved in each storage and reclamation event, and so can track the peer's usage over time.

The grace period attack is discussed by Lillibridge et al [18] in the context of a cooperative backup system. This system provides for challenges between peers, but requires symmetric storage relationships, restricting data placement. They propose two solutions to the grace period attack. In the first, the system must prove itself reliable by storing random data for some time before benefiting from the system. In the second, a centralized third party is used to impose a storage "fine" immediately after a restore request.

FARSITE [2] and Pastiche [7] both identify the need to ensure that peers consume only as much space as they contribute. FARSITE lists quota enforcement as future work. Pastiche lists three potential mechanisms—symmetric storage, solving puzzles [17] to pay for space, and micropayments [4]—but rejects each of them as undesirable.

In addition to these specific storage systems, there have been several efforts to produce a general framework that might be applied to peer-to-peer storage. Fileteller [16] proposes the use of micropayments to account for storage contributed and consumed. Such micropayments can provide the proper incentives for good behavior [14], but must be provisioned and cleared by a third party and require secure identities. Cooper [6] proposes *data trading*, the exchange of equal amounts of data, as a way to ensure fairness and discourage free-loading. However, since all exchanges are of actual data, this requires symmetric storage relationships.

Ngan [21] has proposed a distributed accounting infrastructure. Each peer maintains a signed record of every data object it stores, and every object stored on its behalf elsewhere. Each node periodically chooses another random node to audit. For each file the node claims to hold for some other peer, the auditor retrieves the corresponding peer's record, and compares the two. If the auditor finds a node claiming an object it doesn't actually have, that node can be ejected from the system. This framework differs from Samsara in several ways. First, it requires certified identities to prevent false accusations and to ensure that aliases cannot claim to occupy storage on a node. Second, it tracks only actual usage, not capacity; it detects inflated claims of capacity only when the storage system is nearly fully utilized. Samsara detects inflated claims by any individual node as soon as it exceeds its contribution. Finally, a random audit may catch a cheater, but it is unlikely that the perpetrator claims the auditor itself as a victim. In other words, random

audits benefit the group as a whole, but cost the auditor directly. One could argue that this too is a tragedy of the commons, but on a smaller scale. In contrast, Samsara nodes need to check only that they themselves are not victims; they receive direct benefit by doing so.

Finally, the problem of decentralized resource allocation is beginning to receive significant attention. SHARP [13] is a framework for the secure, distributed allocation and consumption of resources. Like Samsara, SHARP assumes no globally certified identities. However, the mechanisms in SHARP require principals that exchange resource rights to first authenticate one another off-line to establish faith in one another's public keys. Furthermore, resource exchanges in SHARP may be asymmetric, and potentially rely on non-local means to detect and punish nodes that advertise and later withhold services.

7. CONCLUSION

Peer-to-peer storage systems rely on their users to provide cooperative storage. Unfortunately users of these systems are likely to behave selfishly when left to their own devices, storing their own data in the system without providing storage for others. Prior solutions to this problem require mechanisms that run counter to the peer-to-peer ideal: a fully decentralized, self-organizing system imposing little or no administrative overhead.

Samsara enforces fairness in peer-to-peer storage systems without requiring trusted third parties, symmetric storage relationships, monetary payment, or certified identities. Samsara nodes accomplish this by charging peers storage space—via the claims mechanism—in proportion to their consumption. After an exchange, each peer checks the other periodically to ensure that both correctly store the data.

Unresponsive nodes are punished probabilistically. Nodes that suffer transient failures have only to restore a handful of replicas from surviving copies in the system. However, dishonest or chronically unavailable nodes will find that they cannot maintain their data.

The costs of exchange are comparable to current tools for secure file transfer. Space overhead can be reduced if necessary eliminated through claims forwarding. Forwarding increases the exposure to the risk of failure, but such risk is modest under reasonable assumptions of utilization and simultaneous, permanent failure.

Acknowledgments

8. REFERENCES

[1] E. Adar and B. A. Huberman. Free riding on Gnutella. *First Monday*, 5(10), October 2000.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 1–14, Boston, MA, December 2002.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.

[4] M. Blaze, J. Ioannidis, and A. Keromytis. Offline micropayments without trusted hardware. In *Proceedings of the Fifth Annual Conference on Financial Cryptography*, pages 21–40, Cayman Islands, BWI, February 2001.

[5] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 34–43, Santa Clara, CA, June 2000.

[6] B. F. Cooper and H. Garcia-Molina. Peer-to-peer resource trading in a reliable distributed system. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, pages 319–327, Cambridge, MA, March 2002.

[7] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 285–298, Boston, MA, December 2002.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, Banff, Canada, October 2001.

[9] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–54, November 1976.

[10] J. R. Douceur. The Sybil attack. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, pages 251–260, Cambridge, MA, March 2002.

[11] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 247–260, Kiawah Island Resort, SC, December 1999.

[12] P. Ferguson and H. Berkowitz. Network renumbering overview: Why would i want it and what is it anyway? Internet RFC 2071, January 1997.

[13] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[14] P. Golle, K. Leyton-Brown, and I. Mironov. Incentives for sharing in peer-to-peer networks. In *Proceedings of the Third ACM Conference on Electronic Commerce*, pages 264–267, Tampa, FL, October 2001.

[15] G. Hardin. The tragedy of the commons. *Science*, 162:1243–1248, 1968.

[16] J. Ioannidis, S. Ioannidis, A. D. Keromytis, and V. Prevelakis. Fileteller: Paying and getting paid for file storage. In *Proceedings of the Sixth Annual Conference on Financial Cryptography*, pages 282–299, Bermuda, March 2002.

[17] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the Network and Distributed System Security Symposium*, pages 151–165, San Diego, CA, February 1999.

[18] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative Internet backup scheme. In *Proceedings of the USENIX Annual Technical Conference*, pages 29–42, San Antonio, TX, June 2003.

[19] S. Low, F. Paganini, J. Wang, S. Adlakha, and J. Doyle. Dynamics of TCP/RED and a scalable control. In *Proceedings of IEEE/INFOCOM'02*, New York, NY, June 2002.

[20] C. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 117–129, Monterey, CA, January 2002.

[21] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, Heidelberg, Germany, November 2001.

[23] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, Banff, Canada, October 2001.

[24] S. Saroiu, G. P. Krishna, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the SPIE Conference on Multimedia Computing and Networking*, pages 156–170, San Jose, CA, January 2002.

[25] M. Satyanarayanan. *RPC2 User Guide and Reference Manual*. School of Computer Science, Carnegie Mellon University, October 1991.

[26] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference*, pages 149–160, San Diego, CA, August 2001.

[27] T. Ylonen. SSH—Secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, San Jose, CA, July 1996.