Robust Accounting in Decentralized P2P Storage Systems

Ivan Osipkov Peng Wang Nicholas Hopper
Yongdae Kim
Computer Science & Engineering, University of Minnesota
Minneapolis, MN 55455
{osipkov,pwang,hopper,kyd}@cs.umn.edu

Abstract

A peer-to-peer (P2P) storage system allows a network of peer computers to increase the availability of their data by replicating it on other peers in the network. In such networks, a central challenge is preventing "freeloaders", or nodes that use disproportionately more storage on other peers than they contribute to the network. While several existing systems claim to solve this problem, we show that all known approaches are vulnerable to various attacks by either a single greedy peer or a small group of peers. To address this problem, we describe a robust distributed system to account for the storage activities of each peer. We analyze the security of this system, prove that it is secure under a much stronger attack model than previous work, and evaluate the efficiency of a prototype implementation.

1. Introduction

In a peer-to-peer (P2P) storage archival system [12, 19, 7, 1, 2], a peer stores copies of its files at other nodes in the network, so that they can be retrieved in the event of a local file system failure. Such decentralized, P2P storage systems can be an attractive way to increase the availability of data, since they have no central point of failure and any organization can deploy such a system with very little extra cost. However, a central barrier to the widespread adoption of such systems is that current systems rely on the *generosity* of participants in order to work at all. To give a simple example, consider PAST [19], a distributed hash table (DHT) based archiving application which achieves load-balancing and efficient access to stored data. If on average each node in a 100-node PAST system stores 10 GB of data for other nodes, a new node that joins the network and writes 100 GB of files to the network will have to store only about 11 GB for others. If enough nodes behave in this manner, other nodes will be unable to find space for their storage needs and will leave the network. PAST has no mechanism to detect such nodes, and therefore requires peers to blindly trust each other not to cheat in this fashion.

Of course, given that peers store files on such systems to increase availability, *some* level of trust among peers is a central requirement: if one cannot be reasonably certain that in the future other nodes in the network will exist, be online, and provide previously stored files, there is little incentive to use the system. While it may be reasonable to assume that most peers will indeed behave cooperatively, some greedy users may still wish to consume more resources than they provide, degrading the quality of service provided by the system. Thus, the primary goal of this work is to design a low-overhead, distributed P2P accounting system which securely prevents such greedy behavior.

1.1. Research Goals and Challenges

We consider P2P file archiving systems in which every peer plays two roles: Provider — each peer provides part of its local storage to other peers — and Consumer — every peer can store data at other peers. A transaction between peers can be a request to either store or delete a file. After a transaction, one peer gets credit (for donating disk space) and the other one is debited for the same amount. We refer to the difference between the amount of data a peer stores locally for other peers and the amount of data the peer stores at others as its "credit score." A negative credit score means that the peer is contributing less than it consumes. A decentralized P2P accounting system should satisfy the following requirements:

–**Attack-free**: the system should be resistant to abuse by the peers. Namely, a coordinated group of "cheating" (or faulty) nodes may collude to achieve one of two goals: 1) *Freeloading* - where the goal of a coalition of adversarial nodes is to collectively store more in the rest of the network than they store locally for others. 2) *Framing* - a coalition of nodes may collaborate to falsify accounting information of some innocent nodes, *e.g.*, make the credit score of the attacked node from the point of view of the system appear

Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)
0-7695-2540-7/06 $20.00 © 2006 IEEE

to be less than its real value.

While the system may aim to provide confidentiality and integrity of the files stored, we assume that, if necessary, such protections can be implemented orthogonally based on standard cryptographic mechanisms.

–**Usable**: the system should be efficient, scalable and able to handle the dynamic nature of P2P systems such as peer joins and leaves. Furthermore, unlike all other systems of this type (including [12, 6, 19]) file deletion should be supported since typical nodes have finite local storage space.

Meeting these requirements poses several challenges in a decentralized environment. The main challenges are ensuring that information about storage transactions is accurately recorded, and making this information available to other peers upon request. In decentralized systems it is not clear who should record this type of information, why other peers should trust this entity and how peers determine who is storing the required information. The dynamic nature of P2P systems makes the problem even harder since the required information may reside at different peers at different times. Alternative approaches that attempt to discard the need for such accounting suffer other inherent limitations that make them unacceptable for a P2P setting; see Section 2 for further discussion.

1.2. Contribution

The primary contribution of this work is the design, analysis and implementation of a robust, scalable, and secure P2P storage accounting system. This system is secure against a comparatively strong attack model in which a known constant fraction $\epsilon < 1/2$ of all nodes collaborate to circumvent the fairness mechanism, although the protocol overhead increases as ϵ approaches $1/2$. In comparison with previous work, our design allows deletion of archived contents and tolerates nodes which are occasionally offline. We also provide a prototype implementation as well as simulation results showing that the system is scalable and can bootstrap from a small initial set of users.

An important secondary contribution is vulnerability analysis of previous approaches to fair P2P storage systems. In many cases we show that many previous solutions (even those which claim fairness) are generically vulnerable to abuse by small coalitions of peers, usually consisting of a single peer. To our knowledge this is the first analysis to point out these inherent security failures in existing systems.

2. Related Work

Current approaches to fair P2P storage can be roughly classified as either *implicit* or *explicit* accounting schemes. In the following, we refer to the *generic freeloading attack*, in which a peer writes files to the network but maintains the appearance of being offline until he needs to retrieve his files, thus avoiding the need to store files for others.

Implicit accounting approaches attempt to avoid explicitly recording the contribution and consumption of a peer. Typically these systems attempt to maintain the invariant that a peer's credit score is nonnegative by means of direct exchange of storage. Within this class of solutions, we consider *micropayment* schemes and *bartering* schemes. However, in micropayment schemes [4, 13], the central difficulty is token revocation which necessitates presence of online trusted party which is a central point of failure. Bartering approaches eliminate need for such entity by locating peers who are mutually interested in each other's services; in this case a mutually beneficial transaction can take place and the rest of the network need not know anything about it. In particular, a node cannot get away with consuming network resources without adequate contribution. However, the process of finding pairs of mutually interested nodes becomes a critical detail since storage efficiency and scalability depend heavily on finding closely matched nodes quickly. Current research focuses on two different approaches.

One approach, employed by *Samsara* [6] is to have the "non-interested" node force the other node to perform an equitable amount of perhaps futile work such as store some "junk" data. This approach ensures that peers perform some equitable amount of work for the system but wastes network resources, requiring 100% bandwidth overhead and significant storage overhead. Moreover, since the "non-interested" peer does not gain anything from the transaction, it is not clear what its motivation is to participate in the transaction. Samsara attempts to deal with freeloading attacks as follows: when node A storing file F at B goes offline, A's file F is dropped by B with a certain probability p which increases the longer A is offline. If p is large the system may be unusable (e.g., for back-up), but if p is small the adversary may maintain its data in the network without contribution through replication, making it susceptible to the generic freeloading attack.

Alternatively, one can find two mutually interested parties through storage auctions [5]: a peer that wants to store some data polls others asking how much it would have to store in return and chooses the best bid. The authors of [5] are concerned primarily with data preservation and assume trust among the peers. As a result, the proposed scheme cannot be used for general P2P storage networks without being susceptible to the generic freeloading attack. As in Samsara, storage auctions have scalability and usability problems: one must either poll a significant part of the network to achieve the best outcome or limit oneself to a small number of (possibly unacceptable) bids.

In *explicit accounting* systems, a peer may use the services of another without barter but, to guard against abuse, (honest) peers agree not to serve peers that do not ade-

quately contribute to the network. To achieve this, if peer A approaches peer B with a service request, peer B should be able to obtain, efficiently and securely, information on A's storage contribution and consumption. Without a central repository, this information is naturally distributed in the system. Hence, three issues need to be addressed: the location of a peer's information, its correctness and completeness. Current approaches can be categorized based on how they address the first question: 1) peer A presents its usage/contribution to B, 2) peer B polls the network to obtain information on A, 3) peer B obtains information on A from some set of peers assigned to monitor A's activities.

The first approach was used in [14]: to ensure that peers do not misrepresent their information, peers periodically and anonymously (using an anonymous P2P overlay) ping others and check their books. This solution, unfortunately, cannot distinguish in some cases a misbehaving peer from a victimized one and requires arbitration in case misbehavior is detected, making it susceptible to both freeloading and framing attacks. In addition, the suitability of this proposal for file archival is questionable since the proposed solution requires that whenever a peer goes offline its files stored in the network are deleted – if not, the system becomes susceptible to the generic freeloading attack.

In the *polling* approach one obtains information on a particular peer by polling the network [8] or local trusted peers [20]. But, polling approaches generally 1) do not scale to large networks and provide incomplete information making them susceptible to freeloading attacks; 2) do not verify the correctness of poll responses (thus framing attacks are a possibility); 3) are susceptible to collusion attacks in which both peers claim that a transaction took place in order to boost the credit of one peer without the second one suffering consequences; 4) limit applications to those that require only reputation of service providers and not of clients.

In what we dub the *witness approach*, each peer has a fixed set of publicly known witness peers which monitor the transactions of the peer [23, 22]. These witnesses can also be employed to carry out distributed reputation calculations as exemplified by *EigenTrust* [11]. This idea appears in other contexts requiring resource accounting; for example, in the CONFIDANT [3] wireless network project, network neighbors eavesdrop on each other to ensure correct behavior. Having witnesses distributes trust, but the witnesses of a peer can be corrupted with the peer reaping long-term rewards, making the system susceptible to both freeloading and framing attacks by small coalitions.

3. System and Attack Models

We consider a set of computing nodes with sufficient non-volatile storage such as hard disk, connected to a large network such as the Internet, and capable of giving access to local storage over the network. The nodes could be personal computers with an Internet connection. We assume that the nodes are connected to each other using a scalable network overlay, for example, Pastry [18] or Chord [21]. [1] The nodes can communicate with each other reliably using either the underlying network (e.g. Internet with IP addresses) or the overlay (using some required meta-data). In addition, nodes are loosely synchronized. As mentioned previously, each node provides local storage space to other nodes (his *contribution*) and stores his own data at other nodes (his *consumption*). If node A stores a file F for node B, A also provides read-access to the stored file F for B. Each transaction can be either storage or deletion of data.

We assume that the nodes in the system form a *community of common interest*, which has three important consequences. First, all but a few nodes hold a long-term interest in using the system for purposes such as archiving, back-up or simply to increase data availability. As a consequence there is no need to consider attacks to disable the network infrastructure. Second, each node has a public key certificate (PKC) created by some mutually trusted authority. As a result, we assume that each physical member of the network cannot have certificates for multiple identities. *As each node is mapped in one-to-one fashion to a physical identity, from now on we refer to nodes as peers.* Finally, most nodes are willing to do some extra work to contribute to the health of the community, if it has little cost to them; their incentive is the continued availability of the system.

As a result of our "community of common interest" assumption, the class of attacks we consider is significantly narrowed. In particular, the attacks addressed in this paper fall into either the *freeloading* or *framing* categories as explained above. In addition, we assume, as a result of the PKI assumption, that the Sybil [9] attack is hard [2]. A small fraction ϵ of the nodes could be controlled by an adversary to behave either selfish and/or malicious. However, we assume that the adversary can compromise only limited number of nodes in a given time period. Our protocols are most efficient if ϵ is less than 0.1 but can be scaled to accommodate any ϵ up to 0.5 as long as the underlying network and overlay maintain reliable communications.[3] While this assumption may seem strong, we note that many existing protocols are easily abusable by a single peer or a coalition of $O(1)$ peers, regardless of the network size.

Once a file is transfered we assume it is the job of the in-

[1] This is assumed only for implementation convenience, not for security reasons. Any mechanism allowing users to find other peers would suffice.

[2] We note that if possible a Sybil attack can be effective against our system, but this is true of previous work as well: previous systems can be effectively attacked with a constant number of identities, most often 1.

[3] We remark that, when the underlying and overlay networks are not reliable, our protocols can be altered to accommodate this, at a considerable loss of efficiency, and only with a supermajority of honest nodes. We do not describe the necessary modifications here.

terested peer to verify from time to time if the file is actually being stored. Note that because our system supports deletion, the file may simply be deleted if the peer that allegedly stores the file fails to prove that it can access the file.

4. System Design

In our design, each peer has a set of witnesses (chosen from other peers) that monitor its transactions. In essence, this approach allows one to decentralize a previously centralized environment, where the witnesses play the role of the center. Our design has three key innovations to overcome the limitations of previous approaches:

–*Randomly Chosen, Dynamic Witness sets.* In our system, the set of witnesses of a peer change over time; whenever the majority of a peer's witnesses are honest, he cannot freeload, nor can he be framed. We stress that the interesting idea here is not the specific protocol used to achieve this objective, but the *architectural concept* of randomizing a peer's witness set.

–*Storage Expiration.* To correct for the possible occasional damage caused when a majority of peer's witnesses are corrupt, our system requires that a file be "refreshed" periodically so that the information related to its storage is sent to the (new) witnesses that were not informed of the transaction before. By setting the refresh interval, the number of witnesses, and the rate of witness turnover appropriately, we can limit the damage while retaining the simplicity and efficiency of the witness approach.

–*"Formal" Deletion.* To counter the generic freeloading attack, we allow nodes to request that a file stored at an offline node be deleted; the file can then be stored to another node. In this way, nodes that attempt to "freeload" by pretending to accept files and then going offline cannot gain credit from the system for storing these files.

4.1. Protocols

The following notations are used to explain the protocols. See Section 4.2 for details on how anyone can compute the current witness set of any peer.

Symbol	Meaning
$Sign_A(m)$	This contains the message m and signature of m by A
$h(m)$	Hash of message m, where h is a strong hash function
ID_A	ID of A, computed as hash of public key certificate of A
$size(F)$	File size
W_A	A's witnesses
t_c	Current time
$R(W_A)$	Approve/Disapprove reply from A's witness
$X \rightarrow Y : M$	$\forall x \in X, \forall y \in Y$, x sends M to y
t_e	File expiration time
s_w	size of a witness set
$S(A)$	A's signed storage request
$D(A)$	A's signed deletion request
$RR(A)$	A's signed refresh request
$SR(B)$	B's signed receipt

File Storage Protocol Suppose peer A contacts peer B with a request to store file F. Algorithm 1 details the procedure whereby the file and a receipt for storage of this file are exchanged. All the parties involved in the transaction store the receipt along with the file expiration date. This receipt is deleted and the credit values of A and B are adjusted appropriately once the storage request expires.

Algorithm 1 File Storage Protocol

1. Storage Request
$A \rightarrow B : S(A) = Sign_A(storage, ID_A, ID_B, size(F), t_c, t_e, h(h(F)))$
A sends a signed storage request M_1 to B, which contains identities of both peers, file size, current time, file expiration time, and double-hash of file contents.
2. Checking with Witnesses
2-a. $B \rightarrow W_A : S(A)$
B forwards the request to W_A to determine if A would be in compliance with network policies even after B storing the file F.
2-b. $W_A \rightarrow B : R(W_A) = Sign_{W_A}(S(A), Approve/Disapprove)$
Each of W_A checks if the message is fresh by checking the current time t_c contained within $S(A)$. Then, the witness replies to B with signed answer $R(W_A)$. If majority of W_A states that A cannot store the file, transaction aborts.
3. File Transfer
3-a. $B \rightarrow A : Approve$ B informs A that it can send file F to B.
3-b. $A \rightarrow B : F$ Then A sends the file F to B.
4. Signed Storage Receipt
4-a. $B \rightarrow A : SR(B) = Sign_B(S(A), h(F), t_c)$
When file transfer is complete, B computes double-hash of the file and compares it to the value in the storage request $S(A)$ – if these are different, transaction aborts. B sends a signed receipt $SR(B)$ containing $S(A)$, $h(F)$ and timestamp t_c to A. Upon receiving this message, A verifies the freshness, signature and checks if received $h(F)$ is same as hash of the file he sent.
4-b. $B \rightarrow W_B, W_A : SR(B), R(W_A)$, and $W_B \rightarrow A, W_A : SR(B)$
Upon receiving the receipt, W_B credits B and W_A debits A. W_B also forwards the receipt to A and W_A. Note B has incentive to send the receipt to W_B (to claim credits), but not necessarily to W_A (if colluding with A) or A (if trying to frame A by telling A that the transaction aborted claiming, for example, that the hash of the file is wrong.)

Note that the only way for B to obtain credit for the stored file is to send the receipt to W_B, which will send it to W_A who will automatically debit A. In addition, if the transaction is recorded by W_A and W_B, then A receives B's receipt which proves to A that B actually received the file. Also, A cannot be debited unless B receives and stores the file correctly, and B can prove to others that it stored A's file and can collect credits.

File Deletion Protocol The file deletion protocol, shown in Algorithm 2, is one-sided; *i.e.*, peer A above can delete its file F at any time without obtaining B's confirmation (Note that B can be offline). We note that in the protocol, A cannot issue a deletion request for a non-existent transaction. Also, an honest node B has to store the file until a deletion request is issued by A or the storage request expires.

Algorithm 2 File Deletion Protocol

1. Deletion Request:
$A \rightarrow W_A, W_B, B : D(A) = Sign_A(deletion, SR(B), t_c)$
A sends signed deletion request $D(A)$ along with the storage receipt $SR(B)$ of file F to W_A, W_B and B. W_A and W_B look up the receipt in their databases, delete the transaction and adjust credits of A and B, accordingly. B at the same time deletes the file.
2. Distribution of D(A): $W_A \rightarrow B, W_B : D(A)$ and $W_B \rightarrow B : D(A)$
W_A forwards A's deletion request to B and W_B, and W_B forwards it to B. The deletion request is buffered until the file expiration date of the original storage request (if B is offline, it queries W_B for deletion request once it is online again).

Storage Refresh Protocol In the file storage protocol, the stored file F has an expiration date t_e. If A decides to continue storing the file at B and B agrees, the storage transaction needs to be refreshed in order for B to continue obtaining credit for storing the file. If B refuses, A can formally delete the file and store it at a more reliable peer. This refresh protocol is similar to the storage protocol except that the file is not transferred. The refresh protocol is shown in Algorithm 3. Upon completion, every party involved in the transaction replaces the old receipt with the new one. Note that 1) A cannot refresh a non-existent transaction, 2) B can refuse to continue storing the file, 3) at the end of the protocol, A knows if the transaction has been refreshed.

Algorithm 3 Storage Refresh Protocol

1. Refresh Request:
$A \rightarrow B : RR(A) = Sign_A(refresh, S'(A), SR(B), t_c, t_e)$
Peer A sends refresh request containing the receipt $SR(B)$ of the original storage request, updated storage request $S'(A)$, updated current time t_c, and the new expiration time t_e.
2. Checking with Witnesses: Peer B needs to verify if A is allowed to continue storing the file.
2-a. $B \rightarrow W_A : RR(A)$
B forwards the request to W_A to see if A is allowed to continue storing file.
2-b. $W_A \rightarrow B : R(W_A) = Sign_{W_A}(RR(A), Yes/No)$
Each of W_A checks if A is allowed to refresh the file. Then, the witness replies to B with signed answer $R(W_A)$. If majority of W_A states that A cannot store the file, transaction aborts.
3. Signed Storage Receipt
3-a. $B \rightarrow A : SR'(B) = Sign_B(S'(A), h(F), t_c)$
B sends a signed receipt $SR'(B)$ to A. Upon receiving this message, A verifies the signature and message freshness
3-b. $B \rightarrow W_B, W_A : SR'(B), R(W_A)$ and $W_B \rightarrow A, W_A : SR'(B)$
Upon receiving the receipt, W_A and W_B replaces the old receipt with the new receipt. Again, W_B forwards the receipt to A and W_A for the same reason as in the file storage protocol.

4.2. Determining Witnesses

Assume each peer A has a set of witnesses W_A drawn from the network peers. Whenever A wants to participate in a transaction, the witnesses in W_A will be contacted and will maintain A's transaction history. The following properties are required to make this approach secure:

–With high probability (at least 99%), the majority of witnesses in W_A must provide correct information on A.

–Any peer should be able to contact the members of W_A efficiently and securely.

Load balancing concerns lead to the observation that each peer must serve as a witness for some peer, while security requirements dictate that each peer must have several witnesses. To decide which peers serve as witnesses of A, it would be best if for each peer the witnesses are chosen at random and independently, while any peer can determine which nodes are the witnesses of A. For this purpose we use a cryptographic hash function h. Let d denote the number of output bits produced by h. If every peer has m witnesses, one can compute the values $h(i||ID_A), i = 1, ..., m$ which will fall randomly and independently for each peer into the range of values $[0, 2^d - 1]$. If we now map each peer into this range in a random manner by hashing the peer's ID, then the peer that is the next one after $h(i||ID_A)$ can be chosen to be the i-th witness of A.

This approach distributes witness load efficiently and in a balanced manner. Additionally, peer A cannot influence the choice of its witnesses and everyone can determine its witnesses. We still need to determine how many witnesses each peer should have. If everyone has s_w witnesses then every peer is also a witness for (on average) s_w other peers. Reducing s_w will decrease the complexity of any protocols that use the witnesses and reduce the witness burden on peers, while increasing s_w decreases the probability of a corrupt witness majority and thus improves security. A simple calculation shows that when $\epsilon = 10\%$ of nodes are corrupt, $s_w = 5$ witnesses suffice to ensure that less than $\delta = 1\%$ of nodes have corrupt witness majority; in general, a Chernoff bound gives that it suffices to have $s_w = \frac{\ln 100/\delta}{2(1/2-\epsilon/100)^2}$.

Dynamic Witness Change To minimize the effect of collusions our protocols ensure that the witnesses for each peer change with time. For load-balancing reasons, the witnesses should change for each peer at different times and one witness should change at a time. This can be implemented using any of several different cryptographic techniques. We describe here the *Queued Witness Replacement* approach which is implemented by the prototype and provides load-balancing. A different approach, *Random Witness Replacement*, which is simpler conceptually and easier to analyze is given in [16]. In this approach, witness change occurs at the same discrete time-intervals for all peers; and during each witness change, a random choice of current witness is replaced by a random peer. Both protocols, however, 1) maintain the same security guarantees as will be shown later, 2) retain all of the necessary characteristics; in particular, anyone can determine the current witness set of any peer and a peer cannot easily influence the choice of its witnesses, 3) allow for usage of public source of randomness in computations to increase indeterminism. Below, we let PKC_A denote the public-key certificate of peer A and assume that PKC_A contains some randomness not determined by A.

Queued Witness Replacement is specified in Algorithm 4 and computes the set of witnesses for peer A at time t_c.

Algorithm 4 Queued witness set construction

1. On input time t_C (expressed, say, in days) and peer PKC_A, initialize $W_{0,\ldots,s_w-1}(A) := \perp$
2. Initialize $n := 0$
3. For $j \in \{0, \ldots, e-1\}$ do:
(a) if $t_C - \lfloor t_C \rfloor \geq H(j \,||\, PKC_A)/2^k$, set $n := n + 1$.
4. For $i \in \{0, \ldots, s_w - 1\}$, set $W_i(A)$ to the node with ID closest to $H(PKC_A \,||\, H(\lfloor t_C \rfloor - \lfloor (i + e - n)/e \rfloor) \,||\, (i - n) \mod s_w)$

In effect, we keep a queue of s_w witnesses: at random intervals, the oldest witness of A is replaced by a randomly

chosen peer; the intervals are different for each peer A and so witness handoffs are smoothly distributed over time. Let ϕ denote the desired life-span of a witness in days (or another arbitrary time increment). For simplicity assume that ϕ divides s_w and denote $e = s_w/\phi$. Algorithm 4 exhibits the following properties:

–Each day, on average s_w/ϕ witnesses change and each such change happens at a random (but fixed for each interval) time [4]. The schedule for each peer is different.

–The average witness life-span is ϕ days.

–Other peers can determine the current witness set of A autonomously, given PKC_A. A peer cannot easily influence the choice of its witnesses.

Witness Hand-Off The remaining problem that needs to be resolved is how to carry out witness hand-off without propagation of incorrect information. More precisely, consider the following scenario. At one point peer C is able to corrupt a majority of its witness set W_C and manages to store in the network significantly more than its contribution. At handoff time, a new (honest) peer X replaces a witness in W_C: if X keeps the same incorrect information as the faulty majority, the effect of dynamic witness change is minimal.

If all witnesses of W_C collude with C and hide C's remote storage transactions the new witness will not be able to obtain a correct view of C's participation without significant overhead. Thus, our protocol requires peers to specify the expiration time of the transaction; system policy can be used to put an upper bound on the life of storage signatures. Even if a peer is able to corrupt all its current witnesses the effect will last only until the transactions expire.

Suppose a new peer X becomes a witness for C replacing one of the current witnesses. Denote by 1) $RS_D(C)$ the set of storage receipts stored at D, which were either issued by or to C, 2) $DS_D(C)$ the set of deletion requests stored at D, which were either issued by or to C. The protocol for witness hand-off is shown in Algorithm 5.

Algorithm 5 Witness Hand-Off Algorithm

1. X obtains signatures (i.e. storage receipts and deletion requests) relating to C from its current witnesses W_C and C (if it is online).

2. For every current witness D of C and for each (unexpired) storage receipt $s \in RS_D(C)$, X checks $DS_D(C)$ to see if there exists any deletion request related with s (i.e. either issued by or to C). If yes, run the file deletion protocol from Step 2 to delete the file.

3. Optionally, X may ask peers and witnesses associated with (unexpired) storage receipt $s \in RS_D(C)$ if any deletion request was issued to or by them. If so, X runs the file deletion protocol using the deletion request.

4.3. Mechanism Design for Fairness

In the above storage and refresh protocols W_A has to make a decision whether to allow A to store or refresh the specified file size or not. The mechanism enforced by W_A is important to guard the system from abuse. All such mechanisms must follow three important rules:

–W_A must ensure that peers cannot successfully go through the storage or refresh protocol if in the end their contribution to the network falls well below their usage of remote storage. For example, in the above protocols, W_A may decide not to allow peer A to store or refresh if the amount of data it stores for others falls well below its remotely stored data after the transaction.

–The strategy must be balanced between preventing freeloaders and protecting honest peers which have a sudden contribution drop. Peer contribution to the network may fall due to various reasons, e.g., large amount of files stored at a peer are deleted by the file owners and this peer fails to attract new storage requests immediately. The system should allow and/or help the honest peer to recover.

–The system must be able to bootstrap. In particular, a new peer joining the system with no current contribution to the network should be able to start storing at other peers within a reasonable amount of time.

To bootstrap the system, new peers are given some *forward credit* which is set by witnesses to a system-fixed constant and allows peers to store remotely more than what they contribute. The forward credit is computed as follows: when a peer stores more data locally than remotely, its forward credit increases; when a peer under-contributes, his forward credit decreases which guards the system from abuse. In both cases, the magnitude of the change in forward credit increases with the magnitude of the difference between a peer's contribution and consumption. Note that storage expiration ensures that files of peers that under-contribute are eventually deleted and when a peer is absent for sufficiently long time, its files and records held by its witnesses will expire and be purged (only a single signature of this peer needs to be maintained so that when it rejoins it will not longer be given initial credit). Still in case of genuine crash, peer's files will be maintained unless its credit declines (when other peers delete their files at this peer), but even in that case the files will be in the network for some time allowing the peer to retrieve them if it cannot go back online for a long time. The above mechanisms are used in our prototype implementation which is discussed in Section 6. In addition, we provide simulations of the system bootstrapping period.

5. Security Analysis

In this Section, we give brief arguments for the security of the protocols sketched in Section 4. Let us say that the system is in a *correct state* with respect to peer A if the majority of A's witnesses are honest and have an accurate accounting of A's storage consumption and contribution. We

[4] The protocol can be easily modified if we would like to have the same time interval between successive witness changes

first briefly sketch why a peer in such a state cannot freeload and cannot be framed. Next we show that a system in a correct state will stay in a correct state when handoff to an honest witness occurs. Finally we demonstrate that the impact of bad states is minimal by analyzing the fraction of time the system is in a bad state with respect to A and determining, analytically and experimentally, the expected time to recover from entering an incorrect state in terms of the witness turnover and storage refresh intervals.

5.1. Security in a Correct State

Now suppose the system is in a correct state with respect to peer A, and thus peer A has currently contributed more than he has consumed. To invalidate this condition, he must break a security guarantee of one of the request protocols:

–*Storage*: In the storage protocol, when storing at honest peer B, A can only exceed his contribution if more than half of his witnesses W_A sign a request permitting the storage. Since we assume that less than half of the witnesses are corrupt, this means A must forge a signature of at least one honest witness. When B, in correct state, is part of the collusion and tries to store at A, A cannot obtain credit without B being debited unless either all of W_A or majority of W_B are corrupt.

–*Deletion*: A could hope to subvert the deletion protocol by either deleting a file stored at honest node B without notifying B, (so that the file remains accessible while not counting against A) or allowing B to delete a file stored at A without notifying W_A. The former attack is impossible in a correct state, since A must first send the deletion request to W_A and once a majority of W_A sign the deletion request, they will forward it to B. The latter attack is also prevented by the design of the protocol, since B forwards the deletion request to W_A directly. If B, in correct state, is part of the collusion and tries to delete a file at A, the attack succeeds only if majority of W_B *are not* and majority of W_A *are* aware of the deletion, which is prevented since W_A forward deletion to W_B.

–*Refresh*: The attacks here are almost identical to attacks on storage protocol with obvious modifications.

Thus we conclude that in a correct state, the security of the protocol against freeloading is reducible to the security of the underlying signature scheme. We remark that the proof of security against framing attacks with honest witness majority will be essentially the same, since in order for a node B (in correct state) to be framed, it must either receive a forged signature from some honest witness in W_B or some honest witness in W_B must receive a forgery of a signature from B. Notice that this proof of security depends on reliable delivery of messages; if the fraction of attackers is large enough to invalidate this assumption then certain race conditions can be exploited to freeload for the interval in which A has at least one corrupted witness. To prevent such attacks, a secure Byzantine consensus mechanism between witnesses can be implemented, at a significant cost in communication overhead in the resulting system.

5.2. Security of Witness Handoff

We wish to show that an honest witness X receiving the accounting information for user A from W_A will obtain a correct view of A's transactions when for each transaction at least one peer in W_A will have corresponding signatures. Thus a system in a correct state with respect to A will remain in a correct state with high probability. It is relatively straightforward to show that this property is reducible to the security of the underlying signature scheme:

–For each remote storage request issued by A, if at least one current witness has a correct view of the transaction, then X will also obtain a correct view. If the file was deleted at least one witness will have the deletion signature (which is maintained until the storage signature expires). If the file was not deleted and one witness submits C's deletion signature, X will push the deletion through and ensure that the file is deleted by all honest parties.

–A similar analysis applies to local storage receipts generated by A: if at least one honest witness has a record of A's storage receipt, it cannot be discarded; and A's corrupt witnesses cannot generate legal receipts without obtaining signatures from some honest witness of W_B. Thus the ability to increase the apparent local storage of A in witness handoff implies the ability to forge a signature.

–Provided that for each transaction at least one witness has the correct view, all relevant transactions will be available to the new witness.

The case when all current witnesses of A are part of a coordinated collusion is not dealt with in the hand-off protocol. As previously mentioned, such collusions allow A to hide its remote storage. Alleviation of such attacks would be expensive so instead we rely on storage expiration to limit the impact of such attacks.

5.3. Bad State Recovery

It is evident that if a majority of peer A's witnesses become corrupted by A then his consumption may arbitrarily exceed his contribution during the entire period of corruption - the corrupted witnesses simply approve all storage transactions. Likewise, if a majority of a peer's witnesses are corrupted against him, he may be framed, limiting his ability to refresh storage requests for the duration of the corruption. Thus we are interested in two quantities related to such "bad states": the fraction of time that the system is in a bad state with respect to peer A, and the expected time to return to a correct state after entering a bad state.

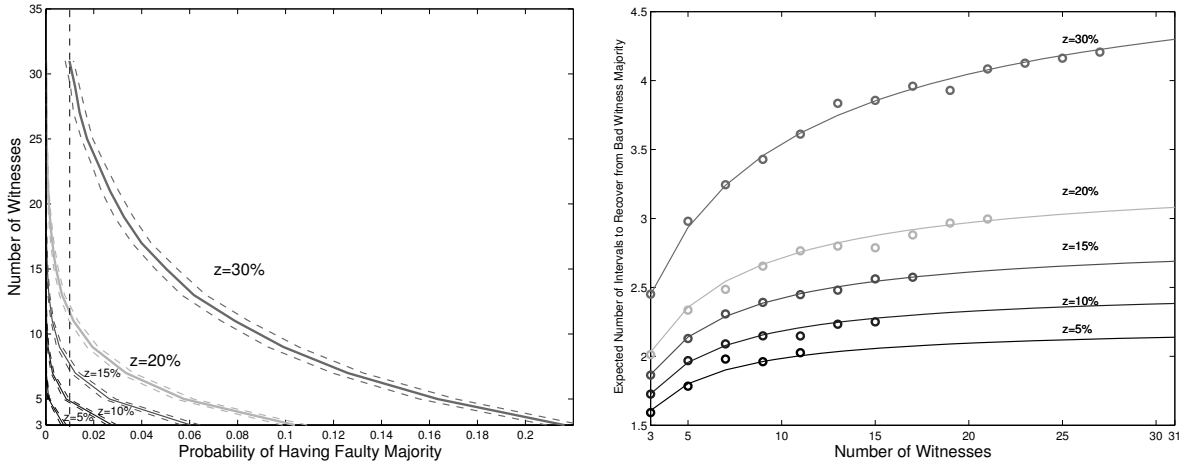

Figure 1. Part (a) shows the expected probability of having faulty majority obtained in simulations using SHA-1 hash function and 10,000 peers: the dashed lines indicate standard deviation. Plot of analytically computed probabilities follows closely this graph. Part (b) shows the average expected time (in terms of witness change intervals) to recover from bad witness majority with z denoting the percentage of bad nodes in the network. The circles show the experimental results obtained using a simulator with the queue-style witness scheduling algorithm over 10000 peers and $k \cdot 10000$ witness changes, where k is the number of witnesses (note that, when probability of bad majority is very low, it takes a long time to obtain accurate results). The solid lines indicate the results obtained using the recurrence relation specified in Section 5.3 and corresponding to the random witness replacement algorithm.

The probability that a random peer ends up with a faulty witness majority can be computed by modeling each witness choice as a Bernoulli trial, where failure probability is equal to the fraction of faulty nodes. The results for cases when this percentage is 5%,10%,15%,20%,30% are presented in Figure 1(a). In particular, in case of 10%, the number of witnesses should be at least 5 (setting used in our implementation in Section 6) to ensure that a random peer ends up with rogue majority with probability less than 1%. Using *Random Witness Replacement* algorithm, one can derive an analytical recurrence-based formula for the expected number of witness hand-offs for the system to recover from rogue witness majority with respect to a single peer. Figure 1(b) shows the results for various numbers of witnesses and percentages of rogue peers: with 5 witnesses and 10% of attackers, the average recovery time, over 5000000 witness changes with 10000 users, was 1.94 with standard deviation 1.1. In addition, the figure shows simulation results with the queued witness schedule of Algorithm 4. Note that the simulation results confirm that both witness replacement approaches exhibit similar expected time to recover from bad witness majority. Once witness majority has recovered, it takes a single refresh interval for them to obtain correct information on the state of the peer.

6. Implementation and Experiments

We have implemented a prototype of the proposed system to determine the amount of computation, communication, and storage overhead imposed by the system, the system scalability and usability. Our implementation is built

Transaction	Storage		Refresh		Delete	
Action	Send	Recv	Send	Recv	Send	Recv
Consumer	2	2+s_w	1	1+s_w	1+2s_w	0
Supplier	2+3s_w	2+s_w	1+3s_w	1+s_w	0	1+2s_w
W_C	1	2+s_w	1	2+s_w	1+s_w	1
W_S	1+s_w	1	1+s_w	1	1	1+s_w

Table 1. The number of messages, where W_C denotes witness of consumer and W_S witness of supplier

in Java on top of the *FreePastry* [10] implementation of the Pastry [18] routing layer. Every peer has an accounting system, a file system, and a Pastry routing layer. The accounting system consists of three modules: the *consumer* module generates storage/refresh/deletion requests; the *supplier* module serves storage/refresh/delete requests issued by other peers; and the *witness* module implements the witness functionality as described in Section 4.

Each module is implemented as a thread with a message queue for incoming messages and a consumer module generates/processes each transaction one at a time. On the other hand, the supplier and witness modules are event driven and are activated when a new message is put in the queue. Each module is equipped with a database in which relevant accounting information such as receipts is stored. The Pastry routing layer is used to locate potential suppliers and contact witnesses. Subsequent communication between the same peers uses direct TCP connections, to minimize communication overhead and dependence on the Pastry routing layer. To speed up cryptographic operations, RSA signatures and verification are implemented using native code written in C with OpenSSL 0.9.8 [15]. To estimate the storage and communication overhead of our system, we recorded vari-

ous messages of the system. The size of the messages varies from 693 to 1229 bytes with an average of 974 bytes. For simplicity, we assume all messages have the size of 1KB. Table 1 shows the number of messages in different transactions. Note that communication overhead per node of the system is linear in s_w. This is a tradeoff between system performance and system security.

Our live testing environment consisted of two different subnets that belong to the Institute of Technology at the University of Minnesota and constitute two separate student labs. The first subnet consists of 44 Sun Blade 1500 machines each with 2GB of RAM and 1062MHz UltraSPARC CPU. The second subnet has 36 machines running Ubuntu Linux (2.6 kernel), with 3GHz Pentium IV CPU and 1GB Ram. The Linux machines have 100Mbit network connection, while the Sun machines are connected with GigE. Transfer of a 100MB file took between 0.8 and 5.2 sec under a light network load. Being a live environment with students using the computers and occasional NFS hiccups, occasional spikes in terms of transaction time do happen, especially in the accelerated experiments described below. Still 95% of all transactions finish within an acceptable amount of time, as indicated in the measurements below.

Although the system is implemented on top of Pastry, to obtain a better view of the overhead induced by our application layer we used a bare-bones Java implementation of DHT routing layer without join/leave mechanisms in the measurements presented here. Figure 3 shows the detailed measurements obtained from our experiments. We started with the following basic setup: 1) 80 nodes with one peer per node, 2) 100 MB files, 3) 5 witnesses for each peer, 4) witness serves for a specific peer for 12 hours (and a witness handoff for a specific peer occurs every 2.4 hours), 5) for each consumer, the average time difference between end of one storage/refresh/delete transaction and start of another is 6 sec. As mentioned before, higher frequency of witness turnover increases security of the system, but increasing it may also affect overall network performance due to more frequent witness handoff. To see how this parameter affects system performance we increased the rate of witness handoff five times (case 80_100_10_1). The figure indicates that the network is able to accommodate such increase of activity without any significant performance penalty. From then on, we used only the setup in which witness hand-off occurs every 2.4 hours. To see how the number of nodes affects performance, we ran the system on 20, 40 and 80 nodes. The results shown indicate that the system scales well as the network size increases. We also ran an experiment using 1 MB files: since length of storage transaction decreases about 5.5 times, the overall rate at which transactions are generated is increased and the results shown indicate graceful degradation of performance of the system.

We also ran a separate simulator, written in C++ with OpenSSL, to determine the the bootstrapping properties of the system [5]. In these experiments, a consumer module issues storage/refresh/deletion requests with equal probability, with the exception that deletion requests become more likely when a peer's consumption exceeds his contribution. The supplier module initially allocates some amount of its local storage for others and grants storage requests if and only if the supplier has free space and the credit of the consumer (including *forward credit*) stays non-negative after the transaction.

We simulated 100 runs with 500 nodes each, with file sizes drawn from a left-skewed distribution with mean 700MB and local allocations uniformly distributed between 10GB and 100GB. The interarrival span of requests was selected uniformly over a small constant range such that each peer made on average two requests per hour, and peers reduced their rate of storage and deletion when consumption reached 95% of local allocation. Maximum file expiration time was set to one day. A more detailed summary of the simulation parameters, along with code for the simulation is available online [17]. Figure 2 shows the bootstrapping behavior when running the simulator. It shows the average of the percentage of local allocation consumed by all peers at each time period. Within 2 days, the system saturates with 90% of users having remotely stored at least 95% of their local allocation and 98.69% of users having remotely stored at least 90% of their local allocation. Thus we conclude that the "forward credit" mechanism effectively allows bootstrapping without deadlock. Note that the system is stable and usable well before the saturation point; on average, a peer has stored 1.1GB in our simulations after 2.74 hours.

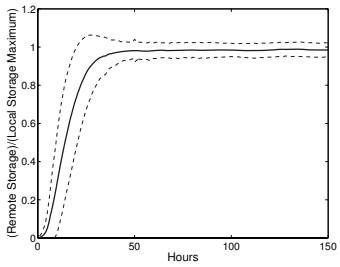

Figure 2. Dynamics of system bootstrap with 500 initial peers.

7. Concluding Remarks and Future Work

In this paper we describe a distributed accounting system for P2P storage archival. Analytical results show that the system is secure against a much stronger attack model than previous fair P2P storage schemes. Our experimental results suggest that the scheme can be efficient even for large networks of peers. However, because there currently are no widely used fair P2P archival systems, we are unable to obtain good traces for experimental purposes. It is easy to see that several usability aspects of our system (e.g., the average wait time to obtain enough credit to write a file) depend on the exact workload. Thus an important next step is

[5] Additional simulations such as "deletion attack" are found in [16]

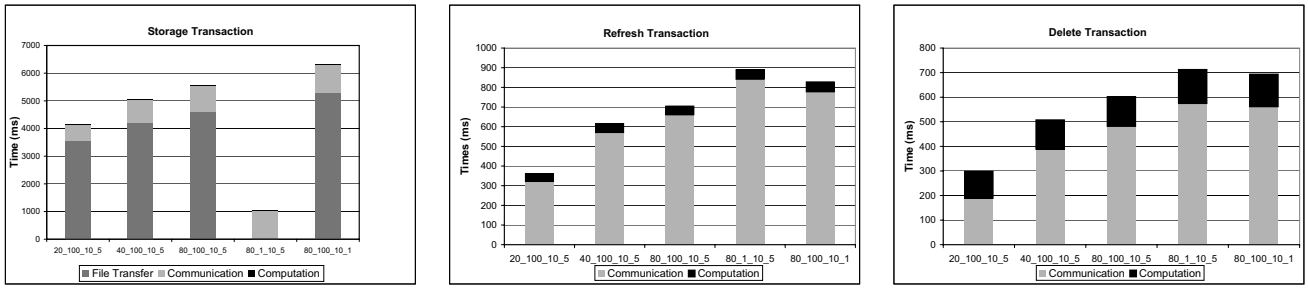

Figure 3. The figure shows average computational, network and file-transfer overhead incurred in the transactions under different experimental conditions. The notation A_B_C_D used on the X-axis means that in the experiment 1) we used A number of nodes, 2) file size was B megabytes, 3) if D=1, the witness hand-off was done 5 times more frequently than in the case D=5. The standard deviation for each measurement ranged up to the value of the corresponding average, due to occasional spikes

to field this system with live users to obtain accurate traces and distribution parameters for simulation.

We have mentioned that the security of the protocols here does not hold against an attacker who can control the delivery of network messages. However, it is possible to modify our protocols to provide assurance in the face of such attacks. An interesting future direction would be to implement such modifications, measure the additional overhead incurred, and find ways to gracefully degrade performance or security when such attacks are detected. Finally, this paper explores one witness mechanism to encourage fairness; in general many other such mechanisms are possible. An interesting future direction is to explore the mechanism design issues related to fair P2P storage, extending the economic concepts introduced in [5].

References

[1] 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 *OSDI*, 2002.

[2] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *ACM SIGMETRICS*, 2000.

[3] S. Buchegger and J.-Y. L. Boudec. Performance Analysis of the CONFIDANT Protocol. In *ACM MobiHOC*, 2002.

[4] B.Yang and H. Garcia-Molina. PPay: Micropayments for Peer-to-Peer Systems. In *ACM CCS*, 2003.

[5] B. F. Cooper and H. Garcia-Molina. Peer-to-Peer Data Preservation through Storage Auctions. In *Transactions on Parallel and Distributed Systems*. IEEE, 2005.

[6] L. P. Cox and B. D. Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. In *ACM SOSP*, 2003.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *ACM SOSP*, 2001.

[8] E. Damiani, D. C. di Vimercati, S. Paraboschi, P. Samarati, and F. Violante. A Reputation-Based Approach for Choosing Reliable Resources in Peer-to-Peer Networks. In *ACM CCS*, 2002.

[9] J. R. Douceur. The Sybil Attack. In *IPTPS*, 2002.

[10] FreePastry Team. Freepastry version 1.4.1. `http://freepastry.rice.edu/`, 2005.

[11] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. In *WWW*, 2003.

[12] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *ASPLOS*, 2000.

[13] R. J. Lipton and R. Ostrovsky. Micro-Payments via Efficient Coin-Flipping. In *FINANCIAL CRYPTO-98*, 1998.

[14] T. Ngan, D. S. Wallach, and P. Druschel. Enforcing Fair Sharing of Peer-to-Peer Resources. In *IPTPS*, 2003.

[15] OpenSSL Project Team. Openssl. `http://www.openssl.org/`, 2005.

[16] I. Osipkov, P. Wang, N. Hopper, and Y. Kim. Robust Accounting in Decentralized P2P Storage Systems. `http://www.cs.umn.edu/~osipkov/accounting.pdf`.

[17] I. Osipkov, P. Wang, N. Hopper, and Y. Kim. Simulator for robust p2p storage accounting. `http://www.cs.umn.edu/research/sclab/Code/metering/`, 2005.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM Middleware*, 2001.

[19] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *ACM SOSP*, 2001.

[20] A. Selcuk, E. Uzun, and M. Pariente. A Reputation-Based Trust Management System for P2P Networks. In *ISOC NDSS*, 2004.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, 2001.

[22] V. Vishnumurthy, S.Chandrakumar, and E. G. Sirer. KARMA : A Secure Economic Framework for Peer-To-Peer Resource Sharing. In *P2PEcon*, 2004.

[23] H. Zhang, D. Dutta, A. Goel, and R. Govindan. The Design of A Distributed Rating Scheme for Peer-to-peer Systems. In *P2PEcon*, 2003.