

# Robust Accounting in Decentralized P2P Storage Systems

Ivan Osipkov, Peng Wang, Nicholas Hopper, and Yongdae Kim

An earlier version of this paper [1] under the same title appeared in the Proceedings of the IEEE 26th International Conference on Distributed Computing Systems (ICDCS), July 2006.

The authors are with the University of Minnesota, Minneapolis.

## 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 introduce a scheme that incorporates robust distributed accounting of the activities of each peer and a distributed control mechanism to provide for fairness while incurring acceptable overhead on large data transfers.

We analyze the security of this base scheme, prove that it is secure under a much stronger attack model than previous work, and evaluate the efficiency of a prototype implementation. Finally, we suggest how to use the base scheme as a building block in conjunction with a probabilistic accounting scheme to design a secure accounting system that is efficient even with data transfers as small as a single file-system block.

## Index Terms

Peer-to-peer systems, decentralized distributed systems, storage systems, distributed accounting.

## I. INTRODUCTION

In a peer-to-peer (P2P) storage archival system [2], [3], [4], [5], [6], 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 attractive for many reasons. Server-based systems require expensive equipment, high bandwidth, dedicated maintenance staff, and competent and trustworthy system administrators. Moreover, physically centralized servers are vulnerable to geographically/politically localized faults and make enticing targets for malicious attacks. On the other hand, decentralized P2P storage systems gather their strength by harnessing the (idle) collective resources of loosely coupled, insecure and unreliable machines. Such systems generally do not require full trust, are self-organizing and consequently require minimal, if any, administrative effort. Also, there is no central point of failure and the machines may span different administrative domains, making them highly resistant to attacks or localized faults. Consequently, such storage systems can be used to provide fault-tolerant data replication, leading to highly

persistent and available data storage. In addition, many such systems are scalable, provide load-balancing and can quickly adapt to changing network conditions.

However, a significant barrier to the widespread adoption of decentralized P2P storage systems is that current systems rely on the *generosity* of participants in order to work at all. To give a simple example, consider CFS [4], a distributed hash table (DHT) based archiving application, which achieves load-balancing and efficient access to stored data, and enforces a quota on how much a peer can write to the network, where a peer  $A$  cannot store on any peer  $B$  more than  $1/N$  fraction of  $B$ 's space, where  $N$  is the size of the network. Assume we have 100 nodes such that, on average, each peer utilizes 15 GB of network storage while at the same time making available 20 GB<sup>1</sup> of its own space to others. Consider the *generic freeloading attack* in which misbehaving peers refuse to store for others altogether, or maintain the appearance of being off-line until they need to access their files (thus avoiding the need to store files for others), while at the same time using network storage and thus depleting network resources. Suppose 25 such misbehaving peers join the network, deciding to each store the maximum allowed amount at other peers<sup>2</sup>. Under the CFS quota system, the greedy users will be able to store a total of 400 GB in the rest of the network, thus leaving 100 GB of free network space. Consequently, honest peers will not, on average, be able to store more than 16 GB in the network and, experiencing inadequate service, may decide to leave it. Without additional security enhancements, such as tamper-proof devices, in systems such as CFS, Ivy and PAST, the network will not be able to function as expected and provide necessary resources to honest peers. This is an issue of fairness: the greedy users are able to get away with using the system without contribution at the expense of others. Were the greedy nodes to provide 20 GB each for the network usage, this situation would not have happened. CFS has no mechanism to detect such nodes, and therefore requires peers to blindly trust each other not to cheat in this fashion. A *fair P2P storage system* should allow peers to use at least an amount of network storage comparable to what they provide to others. If this guarantee is abused by a sufficient number of peers, the guarantee will no longer

<sup>1</sup>If the contributed amount is allowed to vary, it will be much more critical to defend against the attack described here.

<sup>2</sup>Greedy behavior in P2P systems was empirically observed in Gnutella and Napster [7], [8]. Furthermore, some users under-reported their resources to avoid providing services to others [8]. This is also known as the *tragedy of the commons* [9], [10], [11], *i.e.*, the conflict between individual interest and the common good.

hold for others; thus, in a *fair P2P storage system*, users should also not be able to store more in the network than an amount comparable to what they contribute.

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<sup>3</sup>. While it may be reasonable to assume that most peers will indeed behave cooperatively, a few greedy peers may still wish to consume more resources than they provide, degrading the quality of service provided by the system. Moreover, the peer nodes in the system are, in general, vulnerable to attacks and may be compromised by malicious outsiders. In designing fair P2P storage system, our main observation is that, *given a secure accounting system, one can build secure fairness mechanisms provided that the accounting information of any peer can be efficiently and securely retrieved*. If peer *A* wants to store a file at peer *B* and peer *B* can correctly determine if *A* is freeloading by querying *A*'s accounting information, then freeloaders can be barred from using resources at honest peers. Thus, we identify the design of a secure and readily available, low-overhead, distributed accounting system as the primary goal of this paper. Alternative approaches to provide fairness that attempt to discard the need for such accounting suffer other inherent limitations that make them unacceptable for a P2P setting; see Section II for further discussion.

**Requirements** We consider P2P file archiving systems in which every peer plays two roles: Provider — each peer provides local storage space 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 providing 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 secure P2P accounting system* should satisfy the following requirements:

- **Attack-free:** the accounting system should be resistant to abuse by the peers. Namely,

<sup>3</sup>Without any trust assumption, users can still use such systems; however, the data will need to be heavily replicated, consuming bandwidth and storage.

a coordinated group of “cheating” (or faulty) nodes may collude to achieve one of two goals: 1) *Credit Inflation* - where the goal of a coalition of adversarial nodes is to inflate their collective credit; 2) *Credit Deflation* - a coalition of nodes may collaborate to falsify accounting information of some innocent nodes, convincing the rest of the network that the credit score of attacked node is less than its real value. While the system may aim to provide confidentiality and integrity of the files stored, such protections can be implemented orthogonally based on standard cryptographic mechanisms.

- **Usable:** the credit score of any peer should efficiently and securely retrievable upon demand by any network peer. Moreover, the security, trust and accountability mechanisms should be amenable to practical implementation, and must not incur unduly excessive computational and other resource overheads. In other words, 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 [2], [12], [3]) file deletion should be supported, if possible, since typical nodes have limited local storage space.

Meeting these requirements poses several challenges in a decentralized environment. The main challenge is ensuring that credit score of each peer is accurately recorded, and making it available to others 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.

Given an accounting system with the above properties, one can design efficient and secure fairness mechanisms. The goal of such mechanisms is to guard the system against *freeloading* and *framing* attacks, where 1) in a *freeloading* attack a coalition of adversarial peers collude to collectively store more in the rest of the network than they store locally for others (an example is the *generic freeloading attack* introduced above), and 2) in a *framing* attack, the coalition collude to effectively bar an innocent peer from obtaining its fair share of network storage.

**Contributions** The primary contribution of this paper is the design, analysis and implementation of a robust, scalable, and secure P2P storage accounting system and accompanying fairness mechanisms. 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 accounting/fairness mechanisms, although the protocol overhead increases as  $\epsilon$  approaches  $1/2$ . First we provide a *base accounting scheme* designed for archiving of large files, which in comparison with previous work allows deletion of archived contents. Recognizing the need for accounting of transactions as small as a single file-system block, we also sketch how one can use the base scheme and probabilistic accounting to design the *extended accounting scheme*, which allows one to reduce the size of transferred data to a single file-system block without incurring overhead penalty of the base scheme. The extended system is useful when large files have to be split into smaller ones due to fragmentation of network storage or when one wants to use replication techniques such as erasure codes. In addition, it can be readily adapted to P2P file systems such as CFS and Ivy. Unlike related work, both schemes also tolerate nodes which are occasionally off-line. In addition, we provide a performance analysis, prototype implementation as well as simulation results showing that the system is efficient, 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. 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 security weaknesses in existing systems.

## II. RELATED WORK

The most well-known P2P storage systems, such as CFS and Ivy [4], [13] among others, were designed with a weaker accounting model and did not ensure that peers actually stored the data according to system policies, thus leaving a loophole for greedy behavior and attracting peers with inadequate local storage. One exception is PAST [3], which uses tamper-proof devices to enforce fairness. As use of tamper-proof devices significantly restricts applicability of such systems, from now on we discuss fairness mechanisms that do not use such devices.

### A. *Implicit Accounting*

Current approaches to fair P2P storage can be roughly classified as either *implicit* or *explicit* accounting schemes. *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 *micro-payment* schemes and *bartering* schemes. Micro-payment schemes [14], [15], [16] require presence of online trusted party to effect coin/token revocation for double-spending prevention, introducing 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 consume 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 on implicit accounting through bartering focuses on two different approaches. One approach, employed by *Samsara* [12], is to have the supplier force the consumer to store a randomly generated file, called *claim*, of the same size. *Samsara* attempts to deal with freeloading attacks as follows: when node  $A$  storing file  $F$  at  $B$  goes off-line (and therefore  $B$  cannot verify that  $A$  is storing  $B$ 's data),  $A$ 's file  $F$  is dropped by  $B$  with a certain probability  $p$ , which increases the longer  $A$  is off-line. One of the main concerns here lies with the imposed overhead: *Samsara* in its basic form requires 100% bandwidth overhead and significant storage overhead due to the transfer and storage of *claims*. One alleviation, which incidentally also provides some motivation for the storage supplier to participate in the transaction, proposed by the authors is for supplier to transfer to the consumer not a new *claim*, but one of the claims that supplier is already storing for others. The supplier transfers the previously stored *claim* to the consumer and deletes its local copy. As a result, a given *claim* may travel along a chain of peers, thus reducing the storage (not the communication) overhead. However, when a node within the chain fails, the chain is partitioned and files exchanged for this claim will also be dropped down the chain, since none of the peers down the chain who committed to storing the *claim* will be able to prove that they actually store it somewhere. The second concern lies with how probability  $p$  should be chosen. 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

the scheme susceptible to the generic freeloading attack. An alternative to Samsara is to find two mutually interested parties through storage auctions [17]: 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. However, this solution is concerned primarily with data preservation and assumes trust among the peers, making it 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.

### *B. Explicit Accounting*

Alternatively, *explicit accounting* systems allow the supplier  $B$  to determine whether a consumer  $A$  adequately contributes to the rest of the network. Given such an accounting mechanism, fairness can be ensured without barter, provided that (honest) peers agree not to serve peers that do not adequately contribute to the network. The main question that such systems need to address is how peer  $B$  can obtain, efficiently and securely, correct and complete information on  $A$ 's storage contribution and consumption. Current approaches can be categorized based on where peer's information is located: 1) peer  $A$  may itself present its usage/contribution to  $B$ , 2) peer  $B$  may poll 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 [18]: to ensure that peers do not misrepresent their information, peers periodically and anonymously (using an anonymous P2P overlay) ping others and check their *books*. Unfortunately, in some cases, this solution cannot distinguish a misbehaving peer from a victimized one and requires arbitration in case misbehavior is detected, making it susceptible to both freeloading and framing attacks. More precisely, in the proposed system, if storage supplier  $B$  claims that a peer  $A$  has stored a file for  $A$  and during polling  $A$  states otherwise, it is impossible to tell who is telling the truth. Consequently, the best auditor can do is to request that  $B$  deletes the file. Suppose  $B$  was not actually storing the file: in that case, up to the time of detection,  $B$  held credit for this file. Using this observation,  $B$ , by claiming to be storing for others, could appear to be a good citizen without sufficiently contributing to the network. Besides this loophole, this solution may not be suitable for file archival since it requires

that files of off-line peers in the network be deleted – if not, peers may claim to be storing files on behalf of non-existent nodes making the system susceptible to freeloading attacks.

In the *polling* approach one obtains information on a particular peer by polling the network [19] or local trusted peers [20]. The problem with the polling approaches is that, generally, they do not scale to large networks and provide incomplete information making them susceptible to freeloading attacks. Secondly, it is a non-trivial task to verify the correctness of poll responses, thus framing attacks are a possibility. Thirdly, polling approaches are susceptible to collusion attacks generic to reputation systems. For example, when  $C$  wants to store a file, its colluder  $A$  may claim that  $C$  is storing a file for  $A$ . When  $A$  wants to store, the claims of the colluders are reversed. And, finally, they impose overhead on the supplier and not the consumer, thus making them less appropriate for storage applications where it is the consumer’s reputation that needs to be determined. Also note that, due to the polling overhead, some suppliers may be inclined to grant service without thorough verification, which may create a loophole for attacks.

Finally, there is the *witness approach*, in which each peer has a fixed set of publicly known witness peers, which monitor the transactions of the peer [21], [22]. These witnesses can also be employed to carry out distributed reputation calculations as exemplified by *EigenTrust* [23]. This idea appears in other contexts requiring resource accounting; for example, in the CONFIDANT [24] wireless network project, network neighbors eavesdrop on each other to ensure correct behavior. Having witnesses distributes trust, but the witnesses of a peer  $A$  can be corrupted. If the peer  $A$  is in control of its witnesses, it can store any amount of data in the network without contributing, reaping long-term rewards from witness corruption. If another malicious peer  $B$  is in control of  $A$ ’s witnesses,  $B$  can launch a framing attack against  $A$  by having  $A$ ’s witnesses reply to others that  $A$  is greedy. Consequently, having static witnesses gives adversaries long-term incentives for witness corruption.

### III. SYSTEM AND ATTACK MODELS

#### A. Basic Network Assumptions

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. We assume that the nodes are connected to each other using a scalable overlay network, for example, Pastry [25] or Chord [26].<sup>4</sup> 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 time 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  $B$  stores a file  $F$  for node  $A$ ,  $B$  also provides read-access to the stored file  $F$  for  $A$ . Each transaction can be either storage or deletion of data. Once a file is transferred we assume it is the job of the interested peer to verify occasionally if the file is actually being stored. Note that if the peer that allegedly stores the file fails to prove that it can access the file, the systems proposed here allow for the consumer to officially delete the file through explicit deletion request and/or through expiration.

### *B. Community of Common Interest (CCI) System Model*

Although by its nature any P2P system or application entails some degree of cooperation and resource sharing, storage systems require *stronger assumptions and constraints* on the role and behavior of peer nodes. If peers cannot be certain that the nodes storing their files will exist in the future, be accessible and provide stored files, there is little incentive to use the system. Hence such *cooperative P2P systems* have *distinct* characteristics that distinguish them from many general P2P applications such as peer-to-peer file sharing. We formalize these distinct characteristics of cooperative P2P systems as a system model, called *Community of Common Interest (CCI)*. First, all but a few nodes hold a long-term interest in getting the *service* provided by the system (for 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, we assume a relatively stable membership in the system with each node having a public key certificate (PKI) created by some mutually trusted authority. The PKI provides implicit access control and enforces that each physical member of the network cannot have certificates for multiple identities. The nodes could be workstations at a university and the certificate authority could be the department

<sup>4</sup>This is assumed only for implementation convenience, not for security reasons. Any mechanism allowing users to find other peers would suffice.

in charge of maintenance of computing facilities. Another implication of stable membership is a low rate of churn - most nodes are online most of the time. 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.

### C. Attack Model

As a result of our “community of common interest” assumption, the class of attacks we consider is significantly narrowed and fall into either the *credit inflation/freeloading* or *credit deflation/framing* categories as explained in Section I. In particular, we do not consider the Sybil [27] attack <sup>5</sup>, which is difficult due to the above PKI assumption.

To model the fact that, typically, an attacker cannot instantly compromise a target system and systems in general do recover within some time, we introduce a notion of “time periods” and say that in any given time period,  $\epsilon$  fraction of the randomly chosen nodes become vulnerable to compromise; the adversary may then choose among them the nodes he will control for the next time period <sup>6</sup>. The chosen nodes could be controlled by an adversary to behave either selfishly and/or maliciously. We note that the set of vulnerable nodes is not chosen by the adversary. In other words, while we do not consider fully adaptive adversaries, as in theoretical cryptography, we argue that this model is strong enough to capture a significant portion of potential attacks. Our protocols are most efficient if  $\epsilon$  is less than 0.1 but can be scaled to accommodate any  $\epsilon$  up to 0.5 as long as the underlying network and overlay maintain reliable communications.<sup>7</sup> While this assumption may seem strong, we note that many existing protocols are easily abused by a single peer or a constant size peer coalition, regardless of the network size.

<sup>5</sup>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.

<sup>6</sup>In the schemes proposed in this paper, if it takes an adversary sufficiently long time, say one day, to compromise a target system, the state of the network will change enough for such attacks to be equivalent to immediate compromise of a randomly chosen node (of course, provided that the goal is to compromise network infrastructure).

<sup>7</sup>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 super-majority of honest nodes. We do not describe the necessary modifications here.

## IV. SYSTEM DESIGN

In our design, each peer has a set of witnesses (chosen among 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 for a peer. 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, it cannot freeload, nor can it 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 peer’s witnesses.
- *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. This way, the (new) witnesses that were not informed of the transaction before will be notified of the transaction when it is refreshed. By setting the refresh interval, the number of witnesses, and the rate at which witnesses change appropriately, we can limit the damage while retaining the simplicity and efficiency of the witness approach.
- *“Formal” Deletion.* To counter the generic freeloading attack, the files can be deleted either through explicit deletion request or through non-refreshment of storage transaction. When transaction size is on the order of a few file-system blocks, then explicit deletion is replaced by non-refreshment, which avoids deletion overhead at the expense of slightly penalizing consumer.<sup>8</sup> In this way, nodes that attempt to “freeload” by pretending to accept files and then going off-line will not be able to maintain enough credit in order to use the network.

### A. Determining Witnesses

As mentioned before, each peer  $A$  has a set of witnesses  $W_A$  drawn randomly from network peers. Whenever  $A$  wants to perform a transaction, the witnesses in  $W_A$  will participate in the transaction and will maintain  $A$ ’s transaction history. The following properties are required to make this approach secure:

<sup>8</sup>Although explicit deletion immediately gives to consumer back the credit, we believe that saving deletion overhead for small transactions compensates for forcing the consumer to always pay for such transactions until their expiration.

- With high probability (at least 99%), the majority of witnesses in  $W_A$  must provide correct information about  $A$ .
- Any peer should be able to efficiently and correctly contact the members of  $W_A$ .
- The system should be able to recover from targeted attacks against  $W_A$ . In particular, corrupting current witnesses of  $A$  should only have a transient effect. Moreover, it should be hard to determine who will serve as witnesses for  $A$  in the future.

To motivate the need for dynamic witness changes, let us suppose that witnesses of a peer do not change. In that case, once peer  $A$  obtains control (through compromise, bribery *etc*) over current  $W_A$ , it will be able to 1) control its witnesses for as long as the network is functioning, and 2) launch attacks using  $W_A$ , since witnesses  $W_A$  serve as authoritative monitors and information source of  $A$ 's accounting information and behavior. The benefits are long-term and it may be in the interests of  $A$  to spend resources and time on obtaining such control. Alternatively, if a malicious adversary corrupts  $W_A$ , it may be able to launch long-term attacks against  $A$  at the expense of one-time corruption effort. If witnesses of each peer change with time and cannot be determined well ahead of time, targeted witness attacks will have a transient effect and the adversary will have to corrupt new witnesses continuously.

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 can use a cryptographic hash function  $h$ . Let  $d$  denote the number of output bits produced by  $h$ ,  $s_w$  be the number of witnesses assigned to each peer and  $T$  the current time in coarse-grained (e.g., day) increments. Assume that every peer  $A$  has a public key certificate  $PKC_A$  containing random bits not determined by the peer, and set  $ID_A = h(PKC_A) \in [0, 2^d - 1]$ . Then the  $i$ th witness of peer  $A$  at time  $T$  can be the peer whose ID the closest predecessor of  $h(i||T||ID_A)$ ,  $i = 1, \dots, s_w$ . To ensure that witnesses of  $A$  cannot be pre-computed for future times, one can also add a public source of randomness into the hash computation, which will limit the time available for compromise. This approach distributes witness load efficiently and in a balanced manner. Additionally, peer  $A$  cannot influence the

choice of its witnesses. Moreover, everyone can determine its witnesses provided that either network membership is known or some implicit mechanism can be used to determine if a peer with certain ID is the closest successor of a given hash value, *e.g.* using reliable DHT based routing such as Chord or Pastry to contact closest successors of a given hash value. To determine how many witnesses each peer should have, note that reducing  $s_w$  will decrease the complexity of any protocols that use the witnesses and decrease 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  $s_w = \frac{\ln 100/\delta}{2(1/2-\epsilon/100)^2}$ . Finally, note that given that witnesses change over time, a secure *witness hand-off* mechanism is required to transfer peer information to a new witness; however, this mechanism is dependent on the actual information being transferred and will be discussed when we introduce storage protocols.

*Queued Witness Replacement* (implemented in our prototype) is specified in Algorithm 1 and computes the set of witnesses for peer  $A$  at time  $t_c$ , where  $e = s_w/\phi$ , and  $\phi$  denotes the desired life-span of a witness in days (or another arbitrary time increment). For simplicity of presentation, we assume that  $\phi$  divides  $s_w$ . We use a cryptographic hash function  $h : \{0, 1\}^* \rightarrow \{0, 1\}^k$ . In effect, we keep a queue of  $s_w$  witnesses: at random intervals, the oldest witness of  $A$  is replaced with a randomly chosen peer; the intervals are different for each peer  $A$  and so witness changes are smoothly distributed over time. In the Algorithm, 1) each day, on average  $e$  witnesses change and each such change happens at a random (but fixed for each interval) time; the schedule for each peer is different; 2) the average witness life-span is  $\phi$  days; 3) other peers can determine the current witness set of  $A$  autonomously, given  $PKC_A$ , while a peer itself cannot influence the choice of its witnesses.

---

**Algorithm 1** Queued witness set construction

---

1. On input time  $t_c$  (expressed, say, in days) and peer  $PKC_A$ , initialize  $W_{0,\dots,s_w-1}(A) := \perp$ ,  $n := 0$ .
  2. For  $j \in \{0, \dots, e-1\}$  do:
    - (a) if  $t_c - \lfloor t_c \rfloor \geq h(j \parallel PKC_A)/2^k$ , set  $n := n + 1$ .
  3. For  $i \in \{0, \dots, s_w-1\}$ , set  $W_i(A)$  to the node with ID closest successor of  $h(PKC_A \parallel h(\lfloor t_c \rfloor - \lfloor (i+e-n)/e \rfloor) \parallel (i-n) \bmod s_w)$ .
-

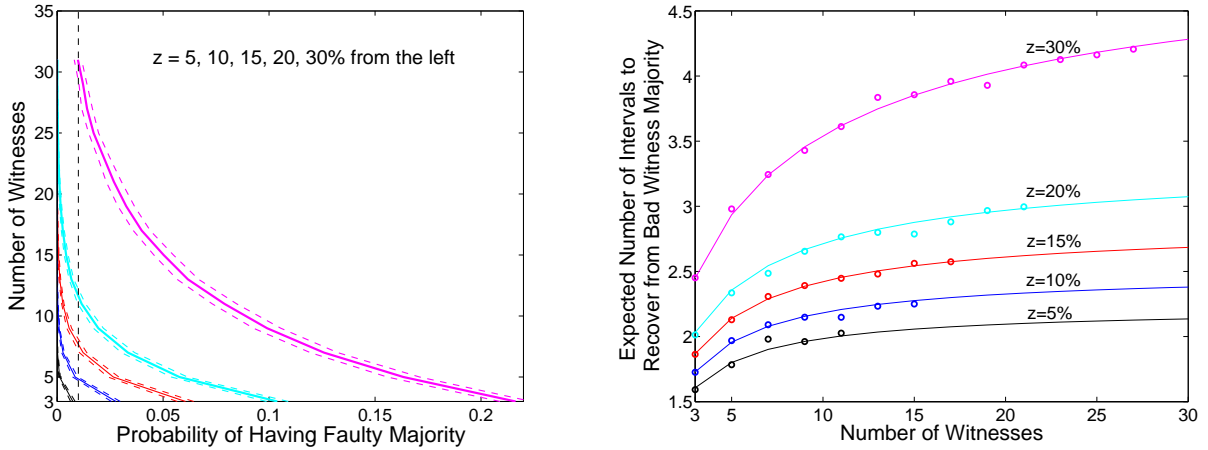


Fig. 1. Part (a) shows the expected probability of having faulty majority obtained in simulations using SHA-1 hash function and 10,000 peers: the dashed lines indicate standard deviation. Plot of analytically computed probabilities follows closely this graph. Part (b) shows the average expected time (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. The solid lines indicate the results obtained using the recurrence relation specified in Appendix A and corresponding to the random witness replacement algorithm.

A different approach, *Random Witness Replacement*, which is simpler conceptually and easier to analyze, is given in Appendix A. 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 with a random peer. Both protocols, however, 1) maintain the same security guarantees, 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.

**Bad State Recovery** We define a “bad state” the situation when majority of witnesses of  $A$  are corrupted. 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.

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  $z$  of faulty nodes. The results for cases when this percentage is  $z = 5, 10, 15, 20, 30\%$

are presented in Figure 1(a). In particular, in case of 10%, the number of witnesses should be at least 5 (the setting used in our implementation in Section VII) to ensure that a random peer ends up in the bad state with probability less than 1%. Using *Random Witness Replacement* algorithm, one can derive an analytical recurrence-based formula, given in Appendix A, for the expected number of witness changes for the system to recover from the bad state with respect to a single peer. Figure 1(b) shows the analytical results for various numbers of witnesses and percentages of bad peers: with 5 witnesses and 10% of attackers, the average recovery time, over 5,000,000 witness changes with 10,000 users, was 1.94 with standard deviation 1.1. In addition, the figure shows simulation results with the queued witness schedule of Algorithm 1. Note that the simulation results confirm that both witness replacement approaches exhibit similar expected time to recover from the bad state.

## V. THE BASE ACCOUNTING SCHEME

The base accounting scheme described in this section allows for three types of transactions: 1) storage protocol is followed when a peer wants to store a file at another peer; 2) deletion protocol is followed when he wants to delete a file; and 3) storage refresh protocol is carried out when he wants to continue storing a file at another peer. The protocols are designed for transactions that involve file-transfer on the order of several megabytes and are suitable to file-archiving schemes where a peer archives its data on a relatively small subset of network peers.

### A. Protocols

The following notations are used to explain the protocols. The witness selection protocols are described in Section IV-A, which shows how to compute the current witness set of any peer.

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

**File Storage Protocol** Algorithm 2 details the procedure whereby the file and a receipt for storage of this file are exchanged, where peer  $A$  contacts peer  $B$  with a request to store file  $F$ . 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. Note that the only way for  $B$  to obtain credit for the stored file is to send the receipt to  $W_B$ , who will forward it to  $W_A$ , resulting in a debit to  $A$ . 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.

---

### Algorithm 2 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  $S(A)$  to  $B$ , which contains identities of both peers, file size, current time, file expiration time, and double-hash of file contents.

#### 2. File Transfer

2-a.  $B \rightarrow A : Approve$   $B$  informs  $A$  that it can send file  $F$  to  $B$ .

2-b.  $A \rightarrow B : F$   $A$  sends the file  $F$  to  $B$ .

#### 3. Signed Storage Receipt

3-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. Otherwise,  $B$  sends a signed receipt  $SR(B)$  containing  $S(A)$ ,  $h(F)$  and time-stamp  $t_c$  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.

3-b.  $B \rightarrow W_B, W_A : SR(B)$ , 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.)

---

**File Deletion Protocol** The file deletion protocol, shown in Algorithm 3, is one-sided; *i.e.*, peer  $A$  can delete its file  $F$  at any time without obtaining  $B$ 's confirmation (note that  $B$  can be off-line). We remark 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 3** File Deletion Protocol

---

**1. Deletion Request:**

$A \rightarrow W_A, W_B, B : D(A) = \text{Sign}_A(\text{deletion}, SR(B), t_c)$

$A$  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 off-line, it queries  $W_B$  for deletion request once it is online again).

---

**Storage Refresh Protocol** The stored file  $F$  has an expiration date  $t_e$ . If  $A$  decides to continue storing the file at peer  $B$  and  $B$  does not object, 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, shown in Algorithm 4, is similar to the storage protocol except that the file is not transferred. 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 4** Storage Refresh Protocol

---

**1. Refresh Request:**

$A \rightarrow B : RR(A) = \text{Sign}_A(\text{refresh}, S'(A), SR(B), t_c, t_e)$

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. Signed Storage Receipt**

2-a.  $B \rightarrow A : SR'(B) = \text{Sign}_B(S'(A), h(F), t_c)$

$B$  sends a signed receipt  $SR'(B)$  to  $A$ . Upon receiving,  $A$  verifies the signature and message freshness.

2-b.  $B \rightarrow W_B, W_A : SR'(B)$  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.

---

**Witness Hand-Off** 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.
- 

*B. Mechanism Design for Fairness*

The above storage and refresh protocols only maintain accounting information of peers and do not enforce any policy. To ensure that greedy peers are not allowed to store, an additional round can be incorporated into the storage and refresh protocols, in which the supplier queries consumer's witnesses to find out if the consumer is allowed to store. The modification required for the storage protocol is shown in Figure 6 (the modification for the refresh protocol is similar).

---

**Algorithm 6** Addition to Storage Protocol

---

**2'. Checking with Witnesses**

2'-a.  $B \rightarrow W_A : S(A)$

$B$  forwards the storage request to  $W_A$  to determine if  $A$  would be in compliance with network policies even after  $B$  has stored the file  $F$ .

2'-b.  $W_A \rightarrow B : R(W_A) = \text{Sign}_{W_A}(S(A), \text{Approve}/\text{Disapprove})$

Each of  $W_A$  checks if the message is fresh by checking the current time  $t_c$  in  $S(A)$ . Then, the witness replies to  $B$  with signed answer  $R(W_A)$ . If majority of  $W_A$  state that  $A$  cannot store the file, transaction aborts.

---

With these additions, the witnesses of  $A$ ,  $W_A$ , now have to make a decision whether to allow  $A$  to store/refresh. Note that  $W_A$  have accounting information of peer  $A$ . However, now one has to decide which policy  $W_A$  should follow when approving/refusing storage requests. The mechanism enforced by  $W_A$  is important to guard the system from abuse and must follow three important rules: 1)  $W_A$  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; 2) the strategy must be balanced between preventing freeloaders and protecting honest peers which have a sudden contribution drop <sup>9</sup>; 3) 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.

We denote by peer *credit* the difference between its contribution and consumption. Generally, consumer is not allowed to store if after transaction its credit becomes negative. However, such policy will not allow the system to bootstrap. To bootstrap the system, new peers can be 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, its 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 under-contributing peers 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 (although one signature should still be kept, so that the peer can not once again obtain the initial credit). Still in case of genuine crash, peer's files will be maintained unless its credit declines (when other peers delete or do not refresh their files at this peer) – however, even in that case, the files will be in the network for some time allowing the peer to retrieve them if it cannot come back online for a long time. The above mechanisms are used in our prototype implementation, which is discussed in Section VII. In addition, we provide simulations of the system bootstrapping period.

### C. Security Analysis

In this Section, we give brief arguments for the security of the protocols sketched in Section IV. 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

<sup>9</sup>Peer contribution to the network may fall for various reasons, e.g., large amount of files stored at a peer are deleted or not refreshed by the file owners and the peer fails to attract new storage requests immediately.

contribution. It is evident that if a majority of  $A$ 's witnesses become corrupted by  $A$  then  $A$ 's credit can be inflated and its consumption may arbitrarily exceed its 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, his credit may be deflated and he may be framed, limiting his ability to refresh storage requests for the duration of the corruption. However, note that the impact of such bad states is minimal: the analytical and experimental results of Section IV show that assuming 10% of corrupted nodes, the expected time to recover from an incorrect state is 1.94 witness changes (with standard deviation 1.1) and once witness majority has recovered, it takes a single refresh interval for them to obtain correct information on the state of the peer. Thus, one can achieve required security by setting rate of witness change and storage refresh interval to appropriate values. Below, we first briefly sketch why in a correct state 1) a peer cannot inflate its credit or freeload, 2) peer's credit cannot be deflated by others and the peer cannot be framed. Next we show that a system in such state will stay in a correct state when hand-off to an honest witness occurs.

*1) Security in a Correct State:* 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, and  $W_A$  have accurate accounting information regarding  $A$ . 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. Thus,  $A$  cannot inflate its credit through storage protocol without another peer  $B$  being debited (and  $B$  actually storing a file at  $A$ ).
- *Deletion:*  $A$  could try 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  $W_A$ , leading to inflation of  $A$ 's credit. 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 the deletion request 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 or credit inflation is reducible to the security of the underlying signature scheme. We remark that the proof of security against framing or credit deflation attacks with honest witness majority will be essentially the same, since in order for a node  $B$  (in correct state) to be framed or have its credit deflated, either 1) some honest witness in  $W_B$  must receive a forgery of a  $B$ 's storage/refresh request, 2)  $B$  should not be aware of deletion of a file stored at  $B$ , or 3) file deletion issued by  $B$  should not be reported to  $W_B$ . Notice that this security proof 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 or inflate credit 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.

2) *Security of Witness Hand-off*: When analyzing security of witness hand-off, the main concern is propagation of incorrect information during witness changes. More precisely, suppose that at one point peer  $C$  manages to corrupt a majority of its witness set  $W_C$  and its accounting information (inflating its credit), and stores in the network significantly more than its contribution. At hand-off 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 storage 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 transactions. Even if a peer is

able to corrupt all its current witnesses, the effect will last only until the transactions expire.

It turns out 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 hand-off 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.

## VI. EXTENSION FOR SMALL FILE TRANSFERS

The base accounting scheme was designed for secure accounting of storage transactions consisting of large (on scale of at least several megabytes) files. However, in many cases, support for transactions that store only a few blocks at a single peer is required. File-systems such as CFS and Ivy [4], [13], for example, use DHash, which replicates each block on several peers based on a random hash key: this approach (a) increases load-balancing, (b) improves robustness/availability

of stored data, (c) provide efficient reads, and (d) improves security against targeted file attacks. In some cases, replication techniques such as erasure codes may be desirable to increase robustness at lower storage overhead compared to simple replication. In others, files will need to be split due to lack of adequate space on any single peer (or a subset of desired peers). Moreover, users may want to append blocks to existing files or delete them. In such scenarios, accounting schemes which can efficiently function with small data sizes are required. The base scheme may not be appropriate for such cases, since it imposes the same overhead independent of data size. As a result, given fixed total data size that a peer wants to store, *the overall overhead of the base scheme increases linearly as the size of transaction decreases*. The goal of this section is to show how to design an accounting scheme where the total overhead is essentially independent of transaction size, thus allowing for efficient storage transactions as small as a single file-system block. Given such an accounting scheme, one can use a variety of fairness mechanisms to ensure that peers contribute to the network, since accurate accounting information is always available to any peer.

Below, we briefly explain the technical details underlying the scheme. The security analysis of the resulting scheme can be easily derived using security properties of the base scheme and the cryptographic properties mentioned below, and is left to the interested reader.

#### A. Main Ideas

Suppose we have a file  $F$  consisting of  $k$  blocks  $F_1, \dots, F_k$ . To minimize overhead of accounting, small transactions either need to be aggregated or only a fraction of them should be reported. In the first case, the transactions may be spread among many peers and aggregation may be fairly expensive. The second case on the other hand requires ability to infer accounting information based on the fraction of reported transactions. We decided to follow the second path using probabilistic approach in which the information about transfer of a given file-block is reported only with certain probability.

To increase performance and reduce use of public key cryptography, we chose to use symmetric-key based cryptography to infer when a file-block is reported. Briefly, on day  $T$  a peer  $B$  computes a symmetric key  $K_{B,T+1}$ , which becomes automatically computable by any other peer on day  $T + 1$ . The schedule of symmetric keys is derived from initial commitment of

peer  $B$  made when it first joined the network and can not be altered. To effect automatic disclosure of  $K_{B,T+1}$  on day  $T$ , we borrow from techniques of timed-release encryption and require existence of a Timed-Release Public Server [28] that periodically (say, once a day) publishes some previously undisclosed information. This server is a general public server similar to NTP servers, which allows for anyone to encrypt messages such that they can be decrypted only starting at some specified date. The full details of leveraging such servers for our purposes are given in Appendix D. Briefly, each peer  $B$  upon joining the network has a certificate that, besides the public key of  $B$ , also has a commitment  $K_B$  such that: 1) given  $K_B$ , date  $T$  and output  $R_{pub}(T)$  of Timed-Release server for this date, anyone can compute the symmetric key  $K_{B,T}$ ; 2)  $B$  can compute  $K_{B,T}$  for any date  $T$ , without requiring any output of Timed-Release server. Note that only supplier can compute its future symmetric keys ahead of time.

Given a file-block on the day  $T$ , supplier decides if the block should be reported by applying a network-fixed  $b$ -bit *keyed message authentication code*, or MAC, with the key  $K_{B,T+1}$  to the block. It outputs a *Hit* when the output is smaller than  $p \cdot (2^b - 1)$  for another network-fixed constant  $p > 0$ , which denotes the probability with which file-blocks are reported. When a supplier  $B$  receives file  $F$ , it applies  $MAC_{K_{B,T+1}}$  to  $F_1, F_2, \dots, F_k$  and reports to the witnesses only the blocks for which *Hit* occurs. When a hit occurs for a block, the supplier follows essentially the protocol of the base scheme for this block, and when there is no hit, the supplier does not report the block. The witnesses, whenever a block is reported, debit consumer for  $1/p$  blocks and credit supplier for the same amount. The consumer requesting a storage space cannot predict the output of the  $MAC_{K_{B,T+1}}$ , while the supplier  $B$  cannot skew the output of  $MAC_{K_{B,T+1}}$  and increase the probability of a hit since the output of the MAC is deterministic and publicly verifiable on day  $T + 1$ . Since all transactions are digitally signed, once the incorrect MAC computation by supplier is detected the next day, the transaction signature proves that supplier has cheated. The keyed MAC is computationally indistinguishable from a pseudo-random function; keyed MAC can be implemented using HMAC – we refer the reader to the Appendix B for formal justifications.

As in the case of all generic probabilistic/sampling schemes, the proposed accounting is by definition only an approximation of the actual amount of stored data. In Appendix C, we discuss

how to set the probability of a hit and the trade-offs involved, and from now on we ignore the issue of correctness of probabilistic accounting.

To ensure that consumers cannot game the suggested probabilistic accounting by re-locating blocks with hits to other suppliers and to motivate suppliers to faithfully store blocks that initially did not produce hits, we disallow explicit deletions and enforce upper and lower bounds on transaction lifetime. The consumer will be charged for any hits until the transaction expires. If the consumer decides to continue storing the block at the given supplier, it will have to refresh the transaction during which the MAC will be re-calculated. The MAC recalculation can be done unilaterally by the supplier at specified periods (if this is allowed, upper bound on transaction lifetime may be set high). To ensure that neither supplier nor the consumer can predict when a hit will occur for a given block, the MAC input may include a public source of randomness such as the one published by Timed-Release server. Thus, supplier cannot predict when a hit will occur for a given block and, on average, will collect its due credit for storing files.

### B. Protocols

Let  $K$  be the transaction size in terms of blocks. Assume for simplicity that  $K$  is a power of two. A *hit* is declared when the  $\sigma$  least significant bits in the MAC are zeros, where  $\sigma$  is a positive integer fixed by the system and will be specified later. Below we introduce the notation used in the protocols:

Symbol	Meaning
$F = F_1    \dots    F_K$	File $F$ consisting of blocks $F_1, \dots, F_K$ .
$R(F)$	Root of Merkle hash tree with hashes of blocks of file $F$ as the leaves
$height_{R(F)}$	Height of the Merkle hash tree with blocks of file $F$ as the leaves
$h(F_i), P_i$	The hash of block $F_i$ and proof that it is part of the Merkle hash tree
$R_{pub}(T)$	The value published by the Timed-Release server on day $T$ .
$K_{B,T+1}$	The secret key used by peer $B$ on date $T$ , which will become available on day $T + 1$ .

The storage protocol is detailed in Figure 7. The Merkle tree is used in order to allow supplier to efficiently and securely prove that a particular block was in fact part of the transferred data.

While, for security reasons, we disallow explicit file deletions, refresh protocol is derived from the storage protocol essentially the same way as in the base scheme. Supplier re-computes

---

**Algorithm 7** File Storage Protocol Using Probabilistic Accounting

---

**1. Storage Request** $A \rightarrow B : S(A) = \text{Sign}_A(\text{storage}, ID_A, ID_B, \text{height}_{R(F)}, h(R(F)), t_c, t_e), F = F_1 || F_2 || \dots || F_K$ 

$A$  sends a signed storage request  $S(A)$  to  $B$ , which contains identities of both peers, current time, file expiration time and  $R(F)$  (and tree height). Peer  $B$  stores hash of each block together with the block.  $A$  sends file  $F$  to  $B$ .

**2. Supplier Computations**

Supplier verifies correctness of storage request and computes HMACs

2-a. (Verification of  $S(A)$ )  $B$  computes  $R(F)$  and verifies  $h(R(F))$  and tree height. Also other properties of storage request are verified (*i.e.*,  $A$ 's signature, freshness, identities). If verification fails, the file is deleted.

2-b. (HMAC Computations)  $B$  computes  $H_i = \text{HMAC}(K_{B,T+1}, h(F_i) || R_{\text{pub}}(T)), i = 1, \dots, m$ . Suppose that file blocks  $F_{i_j}, j = 1, \dots, k$  result in hits. If  $k = 0$ , the transaction ends successfully here.

**3. Signed Storage Receipt**

3-a.  $B \rightarrow A : SR(B) = \text{Sign}_B(S(A), R(F), t_c, h(F_{i_j}), P_{i_j}, j = 1, \dots, k)$

$B$  sends a signed receipt  $SR(B)$  containing  $S(A)$ ,  $R(F)$ , the leaves of the Merkle hash tree that resulted in hits along with the proofs  $P_{i_j}$ , time-stamp  $t_c$  to  $A$ . Upon receiving this message,  $A$  verifies the freshness, signature,  $R(F)$  and proofs.

3-b.  $B \rightarrow W_B, W_A : SR(B)$ , and  $W_B \rightarrow A, W_A : SR(B)$

Upon receiving the receipt,  $W_B$  credits  $B$  and  $W_A$  debits  $A$  (for  $k \cdot 2^\sigma$  blocks).  $W_B$  also forwards the receipt to  $A$  and  $W_A$ . In all cases, witnesses verify proofs  $P_{i_j}$  and  $R(F)$ .

---

each day the keyed MAC of hashes of all blocks that it stores (note that hashes of all blocks are stored along with the blocks). If the file has not expired then the supplier goes through the refresh protocol without involvement of consumer, *i.e.*, unilaterally. If the file expires and consumer wants to continue storing, the refresh protocol would now involve the consumer and the new storage receipt replaces the previous one.

The witness hand-off is identical to the original version with a few exceptions: 1) we no longer have to deal with deletion signatures, which significantly simplifies the protocol, and 2) the new witness may have to perform the maintenance protocol as follows. Each day, the witnesses of each peer  $B$  compute the symmetric key of  $B$  which was used the day before, using the daily output of the Timed-Release server, and verify correctness of keyed MAC computations carried out with  $B$  in the supplier role. Moreover, the witnesses of  $B$  compute the symmetric key of each supplier where  $B$  stored the blocks which resulted in hits, and verify keyed MAC computations of those suppliers as well. Also, each peer  $A$  computes the symmetric keys of its suppliers and verifies keyed MAC computations. In case HMAC values do not result in hits (whereas the supplier claimed that they did), the storage receipt along with supplier certificate constitute undeniable proof that the supplier cheated, since all receipts have time-stamps and signatures.

Transaction	Storage			Refresh			Delete		
	Sig	Ver	Hash	Sig	Ver	Hash	Sig	Ver	Hash
Consumer	1	1	$h(F)$	2	1	0	1	0	0
Supplier	1	1	$h(F)$	1	2	0	0	1	0
$W_C$	0	2	0	0	2	0	0	1	0
$W_S$	0	2	0	0	2	0	0	1	0

Table 1 The computational complexity of the base scheme for each type of transaction, where  $W_C$  denotes witness of consumer and  $W_S$  witness of supplier. Negligible operations are omitted.

## VII. PERFORMANCE

In this section we discuss performance aspects of the proposed schemes. We first theoretically analyze the complexity of the protocols in terms of computation, communication and storage overhead. The following are notable conclusions: 1) in the base scheme, the computational overhead is insignificant compared to the delay introduced by the communication; 2) in the extended scheme, the amortized communication overhead is negligible and the computation becomes more important. Also, since in the extended protocol the witnesses are contacted relatively rarely for small transactions, the total amortized computational overhead is smaller than in the base scheme. Thus, the base scheme is more suitable for large-size transactions in which file transfer delay is dominant and exact accounting is affordable, and the extended scheme works well when transaction size is on the order of 1-1000 blocks.

Note that the communication model in the extended scheme (when witnesses are contacted) is borrowed from the base scheme. To measure the delay introduced by this communication model, we have implemented a prototype of the base scheme: our goal was to measure the system scalability and usability. The implementation and experimental results are presented here. Finally, although the extended scheme does not require bootstrap mechanisms by design, the base scheme does need such mechanisms (see Section V-B). Here we provide simulation results that show that proposed *forward credit* mechanism will allow the system to bootstrap fairly quickly.

### A. Performance Analysis

Tables 1 and 2 show computation and communication complexity of the base scheme. One notable aspect of the protocols is the required communication, *e.g.*, in case of storage/refresh

Transaction	Storage			Refresh			Delete	
	I	II	III	I	II	III	I	II
Consumer	1	0	0	1	0	0	$1+2s_w$	0
Supplier	0	$1+2s_w$	0	0	$1+2s_w$	0	0	0
$W_C$	0	0	0	0	0	0	0	$1+s_w$
$W_S$	0	0	$1+s_w$	0	0	$1+s_w$	0	1

Table 2 The number of messages sent during each round of storage/refresh/delete protocols in the base scheme, where  $W_C$  denotes witness of consumer and  $W_S$  witness of supplier

Transaction	Storage			Auto Refresh			Explicit Refresh		
	Sig	Ver	Hash	Sig	Ver	Hash	Sig	Ver	Hash
Consumer	1	1	2K	0	$\delta^*$	0	1	$\delta^*$	0
Supplier	$\delta^*$	1	4K	$\delta^*$	0	2K	$\delta^*$	1	2K
$W_C$	0	$2\delta^*$	$\delta \cdot \alpha \log K^*$	0	$\delta^*$	$\delta \cdot \alpha \log K^*$	0	$2\delta^*$	$\delta \cdot \alpha \log K^*$
$W_S$	0	$2\delta^*$	$\delta \cdot \alpha \log K^*$	0	$\delta^*$	$\delta \cdot \alpha \log K^*$	0	$2\delta^*$	$\delta \cdot \alpha \log K^*$

Table 3 The computational complexity of the extended scheme for each type of transaction, where  $W_C$  denotes witness of consumer and  $W_S$  witness of supplier. Notation: 1)  $K$  stands for number of file-system blocks in the transaction; 2) asterisk indicates that this is amortized complexity over many such transactions; 3)  $\delta = 1 - (1 - 2^{-\sigma})^K$  is the probability that transaction generates a hit; 4)  $\alpha = 1 + (K - 1)2^{-\sigma}$  is the expected number of blocks that resulted in a hit, given that transaction generates at least one hit. We assume that once computed, the Merkle hash tree is stored. Negligible operations are omitted.

protocols we have three communication rounds and a total of  $2 + 3s_w + s_w^2$  messages sent and in the delete we have two rounds with  $1 + 4s_w + s_w^2$  messages. Noticing that the hash of the file can be computed on-the-fly during the file transfer, the delay imposed by the computational overhead is significantly smaller compared to the communication overhead, as will be corroborated in the experiments below. The computational and communicational complexity of the extended scheme are detailed in Tables 3 and 4. Note that when a hit does not occur during transaction, the witnesses are no longer involved and the transaction finishes once the supplier receives the file. This significantly reduces communication, computational and storage overhead imposed on the witnesses. By the same token, the complexity imposed by the maintenance and witness hand-off protocols is reduced as well.

To give more concrete estimates of overhead for the extended scheme, let us set block size to be 4 KB,  $K = 8$  (32 KB data per transaction),  $\sigma = 12$  and the total number of blocks that each consumer stores  $F = 2^{23}$  (32 GB of data). The Table 5 shows the cost of primitive operations and daily computational complexity imposed on consumer, supplier and witness along with the total daily complexity for each peer. We assumed that each peer has a certificate with its ESIGN

Transaction	Storage			Auto Refresh		Explicit Refresh		
	I	II*	III*	I*	II*	I	II*	III*
Consumer	1	0	0	0	0	1	0	0
Supplier	0	$1+2s_w$	0	$1+2s_w$	0	0	$1+2s_w$	0
$W_C$	0	0	0	0	0	0	0	0
$W_S$	0	0	$1+s_w$	0	$1+s_w$	0	0	$1+s_w$

Table 4 The number of messages sent during each round of storage/ autonomous refresh/explicit refresh protocols in the extended scheme, where  $W_C$  denotes witness of consumer and  $W_S$  witness of supplier. The asterisk next to round number indicates that this round happens with probability  $\delta = 1 - (1 - 2^{-\sigma})^K$  where  $K$  is the number of blocks in a transaction.

1023 public key <sup>10</sup> and signature generation/verification costs were measured using Crypto++ 5.2.1 [29] library. Complexity of hash computations and SHA1-based HMAC implementation of keyed MAC was measured using OpenSSL. Bilinear map complexity was measured using Miracl crypto library [30]. We assumed that every transaction had to be explicitly refreshed on a daily basis, and HMAC verifications are done on a daily basis without any aggregation techniques. To analyze the communication overhead, the size of storage request can be conservatively assumed to be 256 bytes excluding peer certificates, and storage receipt to be 512 bytes each. The total communication overhead induced by single consumer will be about 304,110 KB per 32 GB of transferred data, or about 0.91% of transferred data. Adding conservatively 10 seconds to communication overhead when a hit occurs, the amortized communication delay will be 0.02 seconds per transaction. To analyze storage overhead, note that each consumer and supplier will each have to store about  $F/K \cdot \delta = 2047$  receipts with each receipt about 512 bytes. A peer in witness functionality will have to store  $5 \cdot 2 \cdot 2047$  of receipts. Thus each peer stores about 12,282 KB of receipts. Since receipts from previous time period are retained for delayed verification, the total storage overhead becomes less than 25 MB, or less than 0.08% of stored data.

### B. Implementation, Experiments and Simulations

The implementation of the base scheme was intended to measure the overhead imposed by the communication model in the protocols. Our implementation is in Java on top of the *FreePastry* [31] implementation of the Pastry [25] routing layer. Every peer has an accounting

<sup>10</sup>The alternative RSA signatures can be used as well. However, in this case, signature generation would be several times more expensive. Still note that some operations may be aggregated reducing computational overhead at least twice.

Basic Operations						
Operation	Sig (ESIGN)	Ver (ESIGN)	SHA1 (4KB)	SHA1 (20B)	HMAC (20B)	Bilinear Map
Computational Complexity (sec)	0.00043	0.00016	0.000015	0.00000115	0.0000023	0.018
Daily Peer Overhead						
Functionality	Consumer	Supplier	Witness	Maintenance (Consumer)	Maintenance (Witness)	Total
Computational Complexity (sec)	587	325	6.7	37	184	1138

Table 5 The computational complexity of basic operations and daily complexity imposed on peers. Each peer on average stores/refreshes 32 GB with 8 blocks per transaction and block size 4 KB. Each peer has five witnesses. The basic operations used P4 3.2GHz running Linux 2.6.10.

system consists of three modules: the *consumer*, *supplier* and *witness* implementing corresponding functionalities as described in previous sections. 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. 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, while subsequent communication between the same peers uses direct TCP connections. To speed up cryptographic operations, digital signatures and verification are implemented using native code written in C with OpenSSL 0.9.8 [32]. The size of the messages using Java object serialization varies from 693 to 1229 bytes with an average of 974 bytes.

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

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 for 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 hand-off 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 hand-off. To see how this parameter affects system performance we increased the rate of witness hand-off five times (case 80\_100\_10\_1). The data 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 bootstrapping properties of the system <sup>11</sup>. 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 inter-arrival 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.

<sup>11</sup>Additional simulations such as "deletion attack" are found in [33].

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 [34].

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.

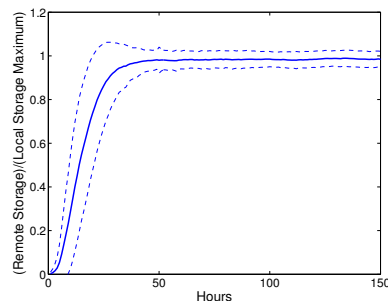


Fig. 2. Dynamics of system bootstrap with 500 initial peers.

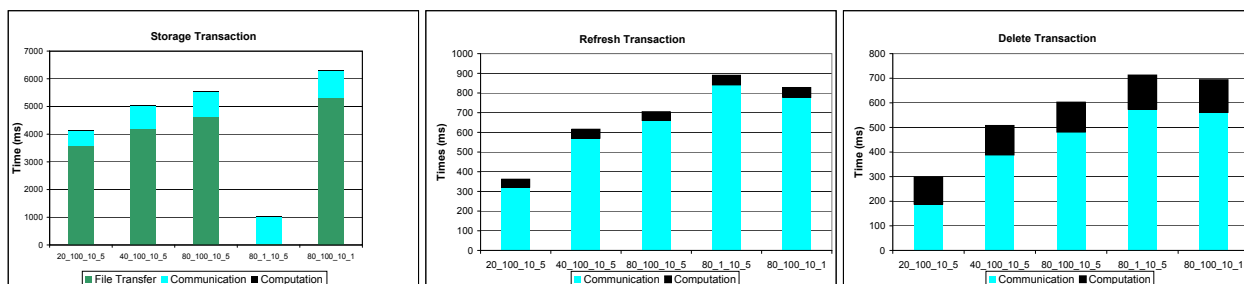


Fig. 3. The figure shows average computational, network, and file-transfer overhead incurred in the transactions under different experimental conditions. The notation A\_B\_C\_D used on the X-axis means that in the experiment 1) we used A number of nodes, 2) file size was B megabytes, 3) if D=1, the 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

## VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we have addressed the issue of robust fairness in decentralized storage schemes. We analyzed the security of previous schemes under collusive attacks and found that most previous schemes in the literature do not maintain fairness in the face of a “generic freeloading attack.” We then introduced, analyzed and implemented a robust scheme for fair distributed storage. The primary advantages of this scheme are:

- **Distributed accounting and allocation.** Our scheme provides accounting and allocation control similar to a centralized entity without the associated central point of failure.
- **Strong probabilistic security guarantees.** Unlike previous schemes, our scheme provably ensures that every peer’s storage allocation is fair in at least a  $1 - \delta$  fraction of time periods, even when the peer is under attack.
- **Low overhead.** The amount of additional bandwidth consumed by our scheme is small compared to the amount of bandwidth and storage used, unlike some recent schemes with 100% overhead.

Our scheme has a novel design in which we use small, rotating sets of peers to securely emulate a central authority. We believe that this design principle – constructing a secure scheme with a central authority and then distributing the work of the authority to rotating “witness” sets – can simplify the design of secure distributed systems for other tasks as well.

## REFERENCES

- [1] I. Osipkov, P. Wang, N. Hopper, and Y. Kim, “Robust Accounting in Decentralized P2P Storage Systems,” in *IEEE ICDCS*, 2006.
- [2] J. Kubiawicz, 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.
- [3] A. Rowstron and P. Druschel, “Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility,” in *ACM SOSP*, 2001.
- [4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, “Wide-Area Cooperative Storage with CFS,” in *ACM SOSP*, 2001.
- [5] 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.
- [6] 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.
- [7] E. Adar and B. A. Huberman, “Free riding on Gnutella,” XeroxPARC, Tech. Rep., Aug. 2000.
- [8] S. Saroiu, P. K. Gummadi, and S. D. Gribble, “A Measurement Study of Peer-to-Peer File Sharing Systems,” in *Multimedia Computing and Networking*, 2002.
- [9] G. Hardin, “The tragedy of the commons,” *Science*, vol. 162, pp. 1243–1248, 1968.
- [10] K. Lai, M. Feldman, I. Stoica, and J. Chuang, “Incentives for Cooperation in Peer-to-Peer Networks,” in *Workshop on Economics of Peer-to-Peer Systems*, 2003.
- [11] M. Feldman, K. Lai, J. Chuang, and I. Stoica, “Quantifying Disincentives in Peer-to-Peer Networks,” in *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, 2003.

- [12] L. P. Cox and B. D. Noble, "Samsara: Honor Among Thieves in Peer-to-Peer Storage," in *ACM SOSP*, 2003.
- [13] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: A Read/Write Peer-to-Peer File System," in *USENIX OSDI*, 2002.
- [14] B. Yang and H. Garcia-Molina, "PPay: Micropayments for Peer-to-Peer Systems," in *ACM CCS*, 2003.
- [15] R. J. Lipton and R. Ostrovsky, "Micro-Payments via Efficient Coin-Flipping," in *Financial Cryptography*, 1998.
- [16] S. Micali and R. R. Rivest, "Micropayments Revisited," in *CT-RSA*, 2002.
- [17] B. F. Cooper and H. Garcia-Molina, "Peer-to-Peer Data Preservation through Storage Auctions," in *Transactions on Parallel and Distributed Systems*. IEEE, 2005.
- [18] T. Ngan, D. S. Wallach, and P. Druschel, "Enforcing Fair Sharing of Peer-to-Peer Resources," in *IPTPS*, 2003.
- [19] 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.
- [20] A. Selcuk, E. Uzun, and M. Pariente, "A Reputation-Based Trust Management System for P2P Networks," in *ISOC NDSS*, 2004.
- [21] H. Zhang, D. Dutta, A. Goel, and R. Govindan, "The Design of A Distributed Rating Scheme for Peer-to-peer Systems," in *P2PEcon*, 2003.
- [22] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer, "KARMA : A Secure Economic Framework for Peer-To-Peer Resource Sharing," in *P2PEcon*, 2004.
- [23] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina, "The EigenTrust Algorithm for Reputation Management in P2P Networks," in *WWW*, 2003.
- [24] S. Buchegger and J.-Y. L. Boudec, "Performance Analysis of the CONFIDANT Protocol," in *ACM MobiHOC*, 2002.
- [25] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," in *IFIP/ACM Middleware*, 2001.
- [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 *ACM SIGCOMM*, 2001.
- [27] J. R. Douceur, "The Sybil Attack," in *IPTPS*, 2002.
- [28] J. H. Cheon, N. Hopper, Y. Kim, and I. Osipkov, "Timed-Release and Key-Insulated Public Key Encryption," in *Financial Cryptography*, 2006.
- [29] "Crypto++ Library 5.2.1," <http://www.eskimo.com/~weidai/cryptlib.html>.
- [30] Shamus Software Ltd., "MIRACL: Multiprecision Integer and Rational Arithmetic C/C++ Library," <http://indigo.ie/~mscott/>.
- [31] FreePastry Team, "Freepastry version 1.4.1," <http://freepastry.rice.edu/>, 2005.
- [32] OpenSSL Project Team, "Openssl," <http://www.openssl.org/>, 2005.
- [33] I. Osipkov, P. Wang, N. Hopper, and Y. Kim, "Robust Accounting in Decentralized P2P Storage Systems (Early Version)," <http://www.cs.umn.edu/~osipkov/accounting.pdf>.
- [34] —, "Simulator for Robust P2P Storage Accounting," <http://www.cs.umn.edu/research/sclab/Code/metering/>, 2005.
- [35] M. Bellare, "New Proofs for NMAC and HMAC: Security Without Collision-Resistance," in *CRYPTO*, 2006.

## APPENDIX

### A. Random Witness Replacement Algorithm

Suppose we had a trusted third party, or TTP, to compute the witness set of peer  $A$ . Then the TTP could assign a random set of  $s_w$  witnesses to each peer initially, and at the start of each hand-off interval release an announcement of the form “At time interval  $t$ , replace peer  $A$ ’s  $i$ th witness with peer  $B$ .” By following these announcements forward, we could clearly construct a witness schedule where at each time period, a random witness is replaced by a random peer. Notice that any peer  $C$  can also reconstruct the current witness set for  $A$  by starting from the most recent announcement and working back in time until each current witness has been announced.

We can use a cryptographic hash function  $h$  to simulate the TTP’s announcement for time period  $t$  by computing  $m_t || w_t = h(t, PKC_A)$ , where  $m_t$  is a 32-bit integer and  $w_t$  is a 128-bit string. This hash can be interpreted as specifying that we should replace peer  $A$ ’s witness  $j = m_t \bmod s_w$ , with the peer  $B$  such that  $ID_B$  is the closest to  $w_t$ . For completeness, we include the code to construct the witness set for  $A$  given  $PKC_A$  and  $t$  in Algorithm 8.

---

#### Algorithm 8 Witness set construction

---

- 1) On input time period  $t_c$  (expressed, say, in days) and peer  $PKC_A$ , initialize  $W_{0,\dots,s_w-1}(A) := \perp$ ,  $k := 0$ ,  $n := 0$ .
  - 2) While  $n < s_w$  do:
    - a) Set  $m || w := h(t - k, PKC_A)$  where  $|m| = 32$ .
    - b) Let  $j = m \bmod s_w$ .
    - c) If  $(W_j(A) = \perp)$  then set  $W_j(A)$  to the peer with ID closest successor of  $w$  and increment  $n$ .
    - d) Increment  $k$ .
- 

We are also interested in bounding the expected length of time the system can be in an incorrect state with respect to a peer  $A$ . Let us denote by  $\tau_w$  the interval between witness hand-offs and by  $\tau_r$  the maximum storage refresh interval. Clearly, in the worst case, once a peer’s witnesses obtain an honest majority, at most one refresh interval is sufficient to recover from any damage caused by a corrupt majority of witnesses. Thus the expected amount of time the system spends in an incorrect state with respect to  $A$  is at most  $\tau_r + X\tau_w$  where  $X$  is the expected number of witness hand-offs to go from a dishonest majority to an honest majority.

To upper bound the expectation  $X$ , let us denote by  $X_i$  the expected number of hand-offs needed to go from having  $i$  honest witnesses to  $i + 1$  for peer  $A$ . If  $w = 2k + 1$  is the number of witnesses for  $A$  and  $\rho$  is the fraction of dishonest users in the network, we can write a recurrence for  $X_i$  as follows, assuming that at each hand-off a random witness is replaced by a random peer as in Algorithm 8:

$$X_i = (1 - i/w)(1 - \rho) + (i/w)(1 - \rho)(1 + X_i) + (1 - i/w)\rho(1 + X_i) + (i/w)\rho(X_{i-1} + X_i + 1) .$$

This can be simplified to  $X_i = \frac{w+i\rho X_{i-1}}{(1-\rho)(w-i)}$ , and since we want to upper bound  $X = X_k$ , and thus  $i/w < 1/2$ , we can upper bound the recurrence by  $X_i \leq 1/(1-i/w-\rho)$  and  $X \leq 1/(\frac{1}{2} + \frac{1}{2w} - \rho)$ . In the case where  $\rho = 0.1$  and  $w = 5$  (the anticipated levels in our implementation) this bound becomes  $2\tau_w + \tau_r$ . Simulations with the queued witness schedule of Algorithm 1 confirm that this bound is a fair approximation: with 5 witnesses and  $\rho = 0.1$ , the average recovery time, over 5000000 witness changes with 10000 users, was 1.94 with standard deviation 1.1. Figure 1 (b) shows the complete simulation results, with each result obtained with 10000 peers and  $k \cdot 10^6$  witness changes where  $k$  is the number of witnesses. In addition, the figure illustrates our analytical bound. Note that the simulation results confirm that both witness replacement approaches exhibit similar expected time to recover from bad witness majority.

### B. Security of Using MAC in the Extended Scheme

Denote by  $MAC_k()$  the MAC with key  $k$ . Suppose we have a set of peers  $\{P_1, \dots, P_n\}$  and each peer  $P_i$  has a random key  $k_i$  that it uses to compute MAC. Consider a query for  $MAC_{k_i}$  with input message  $m$  and consider two types of responses: 1) the responder simply computes  $MAC_{k_i}(m)$ , or 2) the responder outputs a random value (ensuring the same output for the same tuple  $(k_i, m)$ ). In this section we want to show that one cannot computationally distinguish whether random sampling or actual MAC computation was used. In other words, the consumer that stores blocks at suppliers will encounter the same probability distribution of hits as when hits were computed by a single random function.

More precisely, let  $\mathcal{A}$  denote adversary and  $\mathcal{B}$  be the simulator. Define game  $G_n$  as follows:

- $\mathcal{B}$  samples keys  $k_1, \dots, k_n$  at random from the space of MAC keys. Then  $\mathcal{B}$  flips a fair coin

$b$ .

- The adversary queries  $\mathcal{B}$  with pairs of the form  $(i, m)$  where  $i$  denotes that  $MAC_{k_i}$  should be used, and  $m$  is the data input to the MAC. The simulator answers as follows:
  - Case  $b = 0$ :  $\mathcal{B}$  computes  $MAC_{k_i}(m)$  and returns the result.
  - Case  $b = 1$ :  $\mathcal{B}$  samples a random value from the output space of MAC and returns it to the adversary. Repeated pairs  $(i, m)$  return the same results.
- Adversary guesses  $b'$ . If  $b' = b$  we say that  $\mathcal{A}$  wins, otherwise it loses.

We use a simple hybrid argument to first reduce the above game to the one that uses only one peer, *i.e.*, the case when  $n = 1$ . Consider variation of  $G_n$ , called  $G_j$ , in which above the computation of  $MAC_{k_{j+1}}, \dots, MAC_{k_n}$  are replaced by random sampling and denote by  $\epsilon_j$  the advantage in  $G_j$ . In particular,  $G_0$  consists of only random sampling and  $\epsilon_0 = 0$ . Also note that  $|\epsilon_0 - \epsilon_n|$  is the adversarial advantage in the original game above. Let  $|\epsilon_j - \epsilon_{j+1}| > \epsilon/n$ . The difference between games  $G_j$  and  $G_{j+1}$  is that when  $b = 0$  the second game uses  $MAC_{k_{j+1}}$  while the first one uses random sampling. We have  $|Pr[\mathcal{A}_{G_j}(b = 0) = 1] - Pr[\mathcal{A}_{G_j}(b = 1) = 1]| = 2\epsilon_j$  and  $|Pr[\mathcal{A}_{G_{j+1}}(b = 0) = 1] - Pr[\mathcal{A}_{G_{j+1}}(b = 1) = 1]| = 2\epsilon_{j+1}$ . Since  $Pr[\mathcal{A}_{G_j}(b = 1) = 1] = Pr[\mathcal{A}_{G_{j+1}}(b = 1) = 1]$ , we have  $|Pr[\mathcal{A}_{G_j}(b = 0) = 1] - Pr[\mathcal{A}_{G_{j+1}}(b = 0) = 1]| \geq 2\epsilon/n$ . Thus, we can use the adversary to decide if we use  $MAC_{k_{j+1}}$  or random sampling with guessing advantage at least  $\epsilon/n$ .

The above arguments show that one can reduce game  $G_n$  to the game in which a single MAC is used and adversary has to decide if the MAC or random sampling are used. Thus if we can find a family of keyed functions that are secure against this attack, this family can be used securely in our MAC computations. Families of pseudo-random functions exhibit exactly such property and can be used for our purposes. The index of the pseudo-random function can serve as the key. One can implement such function family using HMAC (with SHA1), which was shown to be secure against the above attack [35].

### C. Accuracy of Accounting Using MAC

The reporting to witnesses in the extended scheme is probabilistic, and consequently it may happen that a peer over- or under-pays for its use of network storage and/or its contribution.

Our goal here is to analyze the probability that a peer accounting either 1) (under-payment) allows him to store significantly more than it contributes or conversely 2) (over-payment) does not allow him to store amount comparable to its contribution. More precisely, peer accounting consists of two parts: its contribution and consumption. If both are, say, over-reported the peer still obtains expected utility from the network. The concern lies in cases where the difference between consumption and contribution significantly differs from the actual data.

To carry out such analysis, assume that a hit occurs when the  $\sigma$  least significant bits are all zeros. Then the probability of a hit for a given block is  $2^{-\sigma}$ . Suppose that peer stores a total of  $F$  blocks in the network and other peers combined store the same amount on this peer. Denote by  $C_P$  the probabilistic contribution of the peer and by  $S_P$  its probabilistic consumption as recorded by the accounting system. We would like to find out the probability  $P(\alpha, F, \sigma)$  that  $S_P - C_P \geq \alpha \cdot F$ , *i.e.*, that the accounting error detrimental to the peer constitutes at least fraction  $\alpha$  of network storage used. We can model peer usage of the network with  $F$  Bernoulli trials with success probability  $2^{-\sigma}$ , and the peer contribution can be modeled likewise.

Note that  $P(\alpha, F, \sigma)$  increases when  $\sigma$  increases and decreases when  $F$  increases. In particular, setting  $\sigma = 12$  and  $F = 2^{23}$  (which is 32 GB if block size is 4 KB), the probability that a consumer will actually go over its quota by at least 10% is  $P(0.1, 2^{23}, 12) = 0.067\%$ , while the probability that it will go over by at least 5% is  $P(0.05, 2^{23}, 12) = 5.5\%$ . As  $F$  is halved, the corresponding numbers become 1.2% and 12.8% respectively. However, note that when we fix the over-quota amount to, say, 1.6 GB then the probability that a peer will go over the quota by at least this amount actually increases as  $F$  increases. Depending on system requirements, one can balance the absolute and relative over-quota requirements and set  $\sigma$  appropriately. One thing that needs to be kept in mind is that increasing accuracy of accounting will necessarily result in more hits and higher overhead.

#### *D. Timed-Release Infrastructure*

We assume existence of Timed-Release Public Server, called TiPuS, which allows for timed-release encryption. In effect, TiPuS allows users to encrypt messages for others “into the future”, where the message becomes decryptable only when TiPuS publishes on designated date some verifiable but previously unknown secret. The timed-release public infrastructure is explained in

detail in [28]. In particular, during setup two secure groups  $\mathbb{G}_1$  and  $\mathbb{G}_2$  of prime order  $q$  are chosen along with admissible bilinear map  $e(\cdot, \cdot) : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$ . A public generator  $P \in \mathbb{G}_1$  is chosen and TiPuS publishes authenticated  $s \cdot P$  for some secretly chosen  $s \in \mathbb{Z}_q^*$ . We use additive notation for group operation in  $\mathbb{G}_1$  (where  $a \cdot P$  denotes  $P$  added  $a$  times for  $P \in \mathbb{G}_1, a \in \mathbb{Z}_q$ ) and multiplicative notation for  $\mathbb{G}_2$ . Assuming that decisional Diffie-Hellman problem is hard in  $\mathbb{G}_2$ , and computational Bilinear Diffie-Hellman is hard as well, one can construct a secure timed-release encryption, where TiPuS on date  $T$  publishes value  $R_{pub}(T) = s \cdot H(T)$  ( $H(\cdot)$  is a secure hash function mapping into  $\mathbb{G}_1$ ). Note that  $R_{pub}(T)$  can be publicly verified using standard bilinear map techniques.

In the extended accounting scheme, we do not need the full functionality of the timed-release encryption. What we need is functionality that allows any peer to compute past symmetric keys of any other peer. The idea is to use the fact that the Timed-Release server has chosen a commitment  $s \cdot P$  which creates a schedule of  $R_{pub}(T)$ : the values  $R_{pub}(T)$  can be computed only by TiPuS, but can be verified by anyone using  $s \cdot P$ . Now suppose every peer  $B$  also has chosen some commitment  $K_B$  which creates a schedule  $K_{B,T}$  of symmetric keys such that  $B$  can compute any of the keys. Suppose that the value  $R_{pub}(T)$  published by TiPuS was a back-door for computation of  $K_{B,T}$ , then our goals would be achieved. It turns out that bilinear maps easily allow for such functionalities and below we detail how this can be achieved.

Each peer  $B$  upon joining the network has a public key certificate which also includes its commitment  $K_B = s_B \cdot P$  for some secretly-chosen  $s_B \in \mathbb{Z}_q^*$  known only to  $B$ . To simplify presentation, we assume that  $T$  is incremented using day granularity. Given peer  $B$ , denote  $K_{B,T} = h(e(sP, s_B H(T))) = h(e(s_B P, s H(T)))$  which will serve as the symmetric key for MAC computation during time period  $T - 1$ . Note that once TiPuS has published  $R_{pub}(T)$ , anyone will be able to compute  $K_{B,T}$  using the second formula. However, peer  $B$  can compute  $K_{B,T}$  for any future time without help from TiPuS using the first formula and  $s_B$ . Assuming that TiPuS is trusted, it is a hard problem to compute  $K_{B,T}$  without knowledge of  $R_{pub}(T)$  or  $s_B$ . Also,  $K_{B,T}$  can be viewed as a pseudo-random function, which allows us to use it as the key in MAC computations.

Note that the above simple construction completely eliminates any communication required to

compute previously undisclosed value  $K_{B,T}$  once the  $R_{pub}(T)$  is public. The only requirement is that each peer has to obtain  $R_{pub}(T)$  for date  $T$ : given this single value, any peer can compute symmetric keys used by any other peer on the date  $T - 1$ .