Long term Data Resilience using Opinion Polls

Nikolaos Michalakis[*]
Sun Microsystems Laboratories

Dah-Ming Chiu
Sun Microsystems Laboratories,

David Rosenthal
Sun Microsystems Laboratories

August 23, 2002

Abstract

Opinion Polls can be used as a means to reach weak agreement, an idea introduced by the LOCKSS system [1]. In this paper we investigate a set of protocols that achieve data resilience for the long term using a peer-to-peer network, where mutually untrusted peers are loosely organized. Peers use Opinion Polls to heal corrupted copies of data items instead of conventional methods that use consensus algorithms or cryptography to sign data. We give a brief overview of how LOCKSS performs Opinions, improve the current algorithms and evaluate our protocols in terms of their performance and security against adversary attacks.

1 Introduction

The conventional approach to digital information preservation is based on a central authority. The central authority may be the author, or publisher of the information. In any case, the central authority is the trusted party for providing an authentic copy of the information. To verify whether a given copy is authentic or not does not require the central authority to be online. Instead, a digital signature (of the central authority) can be used by anyone to check the authenticity.

The central authority model faces a number of difficult management problems for long term survivability. The central authority may not survive. More specifically, the keys used to produce the digital signature need repeated updating over time, see [2] for a detailed discussion.

Much work has been done in data replication mechanisms, which are used to provide availability in data access. Data preservation, on the one hand, is a simpler problem - the data does not change, it is only to be preserved. Data resilience is a different problem however because we have to consider in addition to mechanical faults (that corrupt a copy) malicious attacks by adversaries as well.

Many peer-to-peer systems have been proposed recently [3], [4], [5] that by and large, are designed to provide decentralized data storage and access. Resilience is not yet a concern by such efforts.

[*]N.Michalakis is with the Massachusetts Institute of Technology. E-mail contact: nikos@mit.edu. This work was done while an intern at Sun Microsystems.

Recently, the LOCKSS system [1] proposed a decentralized approach for preserving electronic journals by libraries. More abstractly, this approach can be considered for providing long term data resilience for any kind of digital information. In this approach, the trusted central authority is replaced by a large number of peers who all hold a copy of the same data item. Each peer alone is not trust-worthy. Every peer regularly forms *Opinion Polls* and consults with other peers to check the correctness of the local copy of the data item. If we assume the majority of the peers are well-behaving, the question is whether we can trust the opinion of the peer population as an alternative to the central authority.

In the fault-tolerance literature, the classic byzantine generals problem [6], [7] and [8] addresses how a set of peers (processes) can arrive at a consensus in the face of adversary attacks. Group consensus is hard to carry out in a peer-to-peer setting, since at any given time the group membership is hard to enumerate. In addition, group consensus is stronger than what we need. If we have a large number of peers, it is not necessary for them to synchronously ensure all their copies of some data item are correct. The approach of an Opinion Poll taken by LOCKSS is a weaker requirement than a consensus, but in the peer-to-peer setting it can be used to guarantee data resilience over a long time period.

In this paper, we investigate a set of communication protocols in a peer to peer system that can enable mutually untrusted peers to cooperate through Opinion Polls such that they preserve data in a scalable way and for the long term. In Section 2 we set the requirements for the system and describe our assumptions. In Section 3 we provide an overview of the original LOCKSS algorithms. In Sections 4, 5, and 6 we describe our new set of protocols and evaluate how their performance as the network grows and when peers go off-line. In Section 7 we show how our protocols can defend against adversary attacks. Finally, in Section 8 we examine the behavior of the system and its steady state in the long term.

2 Requirements and Assumptions

In this section we describe the requirements under which the system can guarantee data resilience using Opinion Polls. Then we state our assumptions about the network, peers and the adversary. We make our assumptions conservative, by assuming weak peers, powerful adversaries and basic communication mechanisms.

We divide peers with respect to their intentions into two classes: *loyal* peers that follow the protocols and *malicious* peers that do not. With respect to the quality of a specific data item that they store we divide they loyal peers into *healthy* and *corrupted*.

2.1 Requirements

The requirements are the following:

REQ.1 A peer should be able to perform random sampling over the whole peer set to perform unbiased Opinion Polls. That means if the peer knows all the whole peer set it performs pure random sampling on that set. If the peer only knows a pure subset of the peer set then this subset should be updated regularly in an unbiased way such that the effect of performing random sampling on the subset is the same as performing random sampling on the whole set over time.

REQ.2 Votes in an Opinion Poll are seen in public. This means that other peers can observe someone's vote and use that information for either personal feedback for the quality of their data item or to detect attacks.

REQ.3 Opinion Polls should not allow a minority of malicious peers corrupt the data items of healthy peers.

REQ.4 A peer cannot rely on a central authority. It trusts only itself and the opinion of the majority of other peers. Each operates on its own with minimal configuration and management.

REQ.5 A peer cannot rely on cryptography, PKI, digital signatures. Therefore, it can verify the contents of a message only by asking the sender or the receiver of the message. This means it cannot verify the content if the sender and the receiver give different responses.

REQ.6 The system must maintain the property that a large proportion of peers are healthy at all times.[1] How large this proportion should be is evaluated in Section 8.

2.2 Assumptions

The assumptions that we make about the network infrastructure are the following:

NA.1 The network provides a set of uniquely identified network addresses (IP), which serve as peer identification.

NA.2 Each peer can only receive messages destined to its address; by exchanging messages (handshake) a pair of peers can "authenticate" each other using the network address as a weak form of authentication.

NA.3 Each peer can send a message to any other peer in 1 hop, which might consist of multiple network hops. In this case, no other peer sees that message.

NA.3 The underlying network does not necessarily support multicast mechanisms, so peers multicast a message by sending several copies of the same message.

The assumptions that we make about peers are the following:

PA.1 All peers in the network share the same data items. Therefore the view of the peer network is with respect to a single data item. The actual peer network might possibly be larger, but only peers that share the same data item communicate among themselves.

PA.2 A large proportion of the peers are loyal and follow the protocols, in order to keep other peers healthy at all times. Therefore, any random sample of peers is expected to have a large proportion of loyal peers. How large this proportion should be is investigated in detail in Section 8.

[1]By definition, malicious peers are not healthy

PA.3 A peer can be on-line or off-line. After a peer has been off-line for a long period of time it is considered to have exited the system. Peers go off-line independently of each other.

PA.4 A peer has enough memory to store routing state (i.e. other peers) and voting state (such as poll participants and vote contents it has seen). An implementation of the system can set this minimum amount of memory such that all peers comply with it.

The system's adversaries are considered to be all the peers that are malicious. Our assumptions about malicious peers are similar to those by Sit and Morris ([9]). A malicious peer is assumed to be able to do anything, but the following:

AA.1 It cannot convert a loyal peer into a malicious peer by online means.

AA.2 Because of assumption NA.2 it can only receive messages destined to its address, so it cannot intercept messages addressed to another peer. In other words the underlying network infrastructure (routers, firewalls, etc) cannot be controlled by the adversary and are owned by a neutral third party.

To give a flavor of what a malicious peer can do we include the following:

AA.3 It has the capability to forge messages. It can forge the contents and it can spoof any other peer as the originator or the forwarder of a message (including IP addresses).

AA.4 It can send messages to potentially all the other peers.

AA.5 It can know all other malicious peers and communicate with them without the knowledge of loyal peers. Whenever a message reaches a malicious peer it can be assumed it was received by all of them. That implies that whenever a malicious node is invited to participate in a group then it can have the rest of the malicious nodes participate as well. It also implies that malicious nodes can conspire and try to subvert a loyal and healthy peer into a corrupted peer.

AA.6 It can act as a loyal peer whenever it desires to.

3 Lockss Algorithms

We now briefly describe the inter-peer communication algorithm in LOCKSS as well as how it performs Opinion Polls.

There is a group of peers that contains a data item.[2] The population may change overtime, and each peer only has a limited local view of the population. Each peer may become off-line at certain periods of time.

There are three kinds of messages peers use to communicate with each other:

- keep-alive - sent to announce the existence of a peer
- poll - sent to trigger roughly q votes from random peers; q is the quorum size

[2]The group is assumed to have size in the order of 100 peers or higher

- vote - sent to answer a poll; meant to be ”public”, i.e. heard by other voters and monitoring peers

All messages are sent using the same mechanism - hop-controlled flooding. Each peer keeps a list (fixed sized) of friends. When a peer receives a message of hop count $h > 0$, it decrements the hop count and forwards the message to all its neighbors.

How does each peer get its friends list? Initially, each peer is configured with a set of *default peers*. This corresponds to a directed graph consisting of links from regular peers pointing to the small number of default peers. Initial connectivity for the regular peers is very low (can only reach a few peers).

Whenever a peer receives a message, the upstream forwarder is included as a friend (if not already so), and the oldest friend is removed if necessary to keep the friend list size constant. [3] The dynamic updating of the friends list is supposed to help increase connectivity, as well as bias the connectivity towards peers who had recent interaction with the local peer.

Keep-alive messages are sent at regular intervals. This is to ensure that in the absense of polls the on-line peers are connected to each other. Since the offline peers are not around to send keep-alive, they will be pushed off the friends lists over time.

A poll has a desired quorum size, q. Each poll message is sent with an initial hop-count to reach more peers than q. A distributed algorithm inspired by that used in SRM[10] is used to randomly select q voters. Basically, each peer receiving the poll message sets a timer. It is assumed the votes end up reaching most of the other voters. When the timer goes off, if the local peer has not heard more than q votes, it then volunteers itself and sends a vote.

A vote is a message in response to a specific poll. The votes are assumed to reach the poll initiator and most other voters due to the way the friends list is updated (i.e. consisting of mostly the initiator and other voters).

All these messages may be received multiple times, which is the flooding in a richly connected network. This is considered a feature as long as duplication is not too high. Duplicate copies provides some basis to detect messages tampered by forwarding peers.

All the peers who receive enough votes (at least q) will tally the results. There are three possible results:

1. *Almost all the peers agree with the local peer* - this should be the usual case;

2. *Almost all the peers disagree with the local peer* - this should mean the local peer is wrong.

3. *The local peer disagrees with some of the votes and agrees with some other votes* - This in-between situation is not expected to happen. It may be an indication of possible attack by adversaries.

The number of votes sufficient to distinguish these three cases determines what an appropriate quorum size is.

Since all peers monitor polls initiated by others, the overall rate of polls can be kept at roughly a constant.

[3]With some probability the originator of a message is also included in the friends list.

As we will demonstrate later with simulation, the LOCKSS algorithm does not perform very well. Due to the constant update of the friends list to include recent senders, the global connectivity is rather low - this affects the randomness of the peers the poll messages reach. Also, duplicate delivery is highly uneven.

In practice, LOCKSS still worked reasonably well because it uses multicast whenever it is available and in the experimental system, part of the network had multicast support.

4 Problem Decomposition and Approach

In order to perform Opinion Polls reliably we identify the following subproblems that our protocols have to solve:

Connectivity: Every peer should be able to reach any other peer in the network. Over a long period of time every peer should have heard from or send an advertisement to every other peer in the network. A solution to this problem is necessary to reach requirement REQ.1.

Random Sampling: Every peer should be able to randomly select peers from the whole peer set. The solution to this problem satisfies requirement REQ.1

Poll setup: A peer should be able to initiate a poll and organize with other peers such that they can send and receive votes reliably and the initiator can draw a definite conclusion. A solution to this problem helps satisfy requirement REQ.3 and REQ.4.

Vote Monitoring: Poll Participants can see multiple copies of every vote they observe, such that they can draw conclusions for the quality of the poll and the quality of their data items. A solution to this problem satisfies requirement REQ.2, REQ.5 and in combination with a solution to the Poll Setup problem satisfies requirement REQ.3.

Solutions to all the above problems together satisfy requirement REQ.7.

We introduce the Friend Discovery Protocol that deals with the Connectivity and Random Sampling problems. Using that protocol every peer maintains an equal number of links. A network that is created when each peer has an equal number of links to other peers is called an *exponential* network. Such networks are robust against attacks as presented in [11] and [12]. The Friend Discovery Protocol is described in detail in Section 5.

We introduce the Poll Protocol that deals with the Poll Setup and the Vote Monitoring problem. Using that protocol every peer can perform Opinion Polls in an unbiased way. The Poll Protocol is described in detail in Section 6.

Like in LOCKSS our protocols use keep-alive, poll and vote messages. Details and modifications on each type of message are described under the corresponding protocols.

To evaluate the performance of our protocols we have created a round based simulation of our peer network using the Ascape agent modeling tool [13]. Each peer in the system was represented as an agent. with an ID from 0 to $number\_of\_peers-1$. Each agent was designed to perform the following actions every round:

- perform a check and set the on-line, off-line status. So, with some probability an agent might go off-line.
- if the agent is online then it participates in the friend discovery protocol and the poll protocol.

Peer 0 was the only peer designed to initiate polls for simplicity purposes.

Using the simulation we examined how the network connectivity[4] is improved compared to the LOCKSS algorithms. We further examined how the connectivity degrades as more peers are added while keeping a constant number of links per peer. Then, we examined how friend discovery and polls are affected as peers go off-line randomly while the system is in operation.

With respect to metrics, we observed the fraction of peers unreachable, U_m, when a keep-alive message is flooded, and the *global* and *local* efficiencies of the network, E_{glob} and E_{loc} respectively, two quantities introduced by [14].

We represent our network as a directed graph G with the peers as the nodes and the links from a peer to its friends in the friend list. We denote as N the number of peers in the network.

U_m is a good indication of how well a message gets flooded in the network. It is defined as $U_m = \frac{|R_m|}{N}$, where R_m is the set of peers that received the message m.

As in [14] $\epsilon_{ij} = \frac{1}{d_{ij}}$ is the efficiency between peer i and j where d_{ij} is the shortest distance in hops in G from node i to node j. We use efficiency instead of distance because if a graph is disconnected and nodes i and j are in different connected components then $d_{ij} = \infty$.

Global efficiency gives us a good idea of how many hops peers are away from each other. It is a good indicator of how easily peers can get discovered by other peers in the Friend Discovery Protocol. It is defined as:

$$E_{glob} = \frac{1}{N(N-1)} \sum_{\forall i,j \in G i \neq j} \epsilon_{ij}$$

.

In order to get an idea how E_{glob} is related to the expected shortest distance $E[dij]$ we use the well known inequality between the harmonic mean H and arithmetic mean A of a set of positive numbers. That is $G \leq A$. $A = \frac{a_1+a_2+\ldots+a_n}{n}$ and $\frac{1}{H} = \frac{\frac{1}{a_1}+\frac{1}{a_2}+\ldots+\frac{1}{a_n}}{n}$. Defining our set as the distances d_{ij} we get that $E_{glob} = \frac{1}{H}$ and $E[d_{ij}] = A$ so $\frac{1}{E_{glob}} \leq E[d_{ij}]$. so $\frac{1}{E_{glob}}$ is a lower bound for the average shortest distance between two peers in the network.

Local efficiency is a measure of how well connected are the peers that a peer directly points to. The more links between them the higher the local efficiency. A complete graph has perfect local efficiency since all possible paths are there.

We define as G_k the subgraph that consists of the neighbors of node k. We denote as N_k the number of nodes of G_k.

$$E_{loc} = \frac{1}{N} \sum_{\forall k \in G} E_{loc_k}$$

where

[4]We loosely define connectivity as the ability to for a peer to reach another peer over time given a a certain connection topology

$$E_{loc_k} = \frac{1}{N_k(N_k-1)} \sum_{\forall i,j \in G_k i \neq j} \epsilon_{ij}$$

In other words E_{loc_k} is the global efficiency of the subgraph G_k.

5 Friend Discovery

The friend discovery protocol helps peers in learning about new peers and setting up their routing information. Similarly to LOCKSS, peers enter the network using a bootstrapping mechanism. After they are connected they periodically advertise their existence by sending keep-alive messages. At the same time, they listen to incoming keep-alive messages and record peers in *memory*. Every peer keeps a friend list to whom it forwards keep-alive messages just like in LOCKSS. However, unlike LOCKSS the friend list is updated from entries in memory.

5.1 Memory

A certain amount of memory is necessary in order to perform message routing and voting. The following memory models are attempts to satisfy requirement REQ.1 and help satisfy requirement REQ.2.

5.1.1 Infinite Memory Model

A peer with infinite memory remembers all the peers it has heard messages from, therefore:

1. It can perform perfect random sampling, thus satisfying requirement REQ.2.
2. It automatically satisfies requirement REQ.1.

5.1.2 Filtering Model

If a peer has limited memory, it can remember only a constant number of peers at any given time. These entries are stored in a fixed size list called the *buffer list*. The size of the buffer list has to be at least as the minimal required memory as defined by requirement REQ.1.

A peer has to periodically refresh its buffer list with new entries in an unbiased way so that it can perform random sampling and fulfill requirement REQ.2. It listens to advertisements by other peers and it uses a filtering mechanism to select the new entries. This is described in more detail in Section 5.4.

5.1.3 Filtering with Large Memory Model

If the peer combines the two memory models described above it gets a large buffer list, called the *reference list*, and at the same time uses a filtering mechanism to update list entries. The reference list is large enough to approximate the properties of the infinite list even if the number of peers in the system is very large. It fulfills requirement REQ.1 and it has a large enough number of entries to allow a peer to fulfill requirement REQ.2 without any elaborate filtering mechanism. We assume that the reference list can hold a number of entries much larger than the number of malicious peers in the network.

At first glance, this looks like it might not be an easily applicable model for a peer's memory, but let's present a few examples that demonstrate otherwise:

- A normal PC these days can easily devote 500MB of data of its hard drive to store a file that contains 500 byte entries for a million other peers, where each entry could include an ID, IP address, a reputation and some comments for example. Such a list can approximate an infinite list for about 100 different data items if we assume that at most 10000 peers hold the same data item.

- A MICA wireless sensor board [15] has 128KB of flash memory and it could save from 100 up to maybe 1000 entries depending on how many bytes it should allocate for each peer entry. That means that even if we deploy 1000 sensors that could talk to each other, each sensor has enough entries in its list to approximate an infinite list.

5.2 Keep-Alive Message

Keep alive messages contain the originator of the message and the sender (forwarder) of the message. They also contain a hop count value that every forwarder decrements before forwarding the message. Extra information that could be attached to this message such as the full list of peers that this message traversed before reaching someone is not currently used by our protocol, but maybe it could optimize friend discovery by providing more peers to be added to memory every time a message is received.

5.3 Bootstrapping

We do not enforce a particular bootstrapping mechanism, although we believe certain mechanisms have security advantages over others, while others are more easily applicable. The two most common ones are those that use a *default list* or a *trusted friend*. In the first each peer knows a small number of peers that can contact in order to enter the network. This assumes that these peers are online and working properly. Denial of service attacks on the default list could cause serious problems. On the other hand, a trusted friend is a peer that the incoming peer has built offline trust with and it contacts to enter the network. This mechanism is highly decentralized and less prone to denial of service attacks. This might work well in a corporate or academic environment. However it is not easily available to other environments where it is difficult to build offline trust.

5.4 Advertising and Listening

The algorithm for advertising that we use is the following: The peer generates a keep-alive message with a fixed hop count. It then forwards the message to each of its friends in the friend list. Every friend forwards the message to its own friend list excluding the sender if it is in its friend list. The forwarding continues until the hop count goes to zero.

Every time a peer receives a keep-alive message and it decides to record the message it will add to memory both the sender and the originator of the message.

The algorithm for listening depends on the memory model of each peer. If the peer has an infinite list or a partially filled reference list then it listens to all incoming messages and adds both the sender and the originator in its memory. If the peer has finite memory it should then have a policy on how it replaces entries when its memory is full, so that it does not fill its list with malicious peers in the long term. We assume all peers have a reference list, so when a list is full the peer can select a random peer from the reference list and replace it with a peer from a newly received message if that peer does not already exist in the list. A much more elaborate policy needs to be done if a peer has a buffer list and currently we have not explored in detail a policy that we are confident will avoid attacks from malicious peers. We leave this as future work.

5.5 Updating the friend list

Periodically every peer refreshes its friend list using the information in its memory. The algorithm for updating the friend list is to perform random sampling from the entries in memory, so that links to immediate neighbors dynamically change in an unbiased way. That guarantees that advertisements will get flooded in an unbiased way. If this list is not updated in a unbiased manner then an adversary can perform several attacks as described in Section 7.

5.6 Evaluation

The advertising and listening algorithms provide a solution to the Connectivity problem in combination with the friend list updating algorithm. Furthermore, they help solve the Random Sampling problem for peers with buffer lists.

For the Friend Discovery Protocol E_{glob} and U_m are the two metrics that are relevant to evaluate the performance of the algorithms.

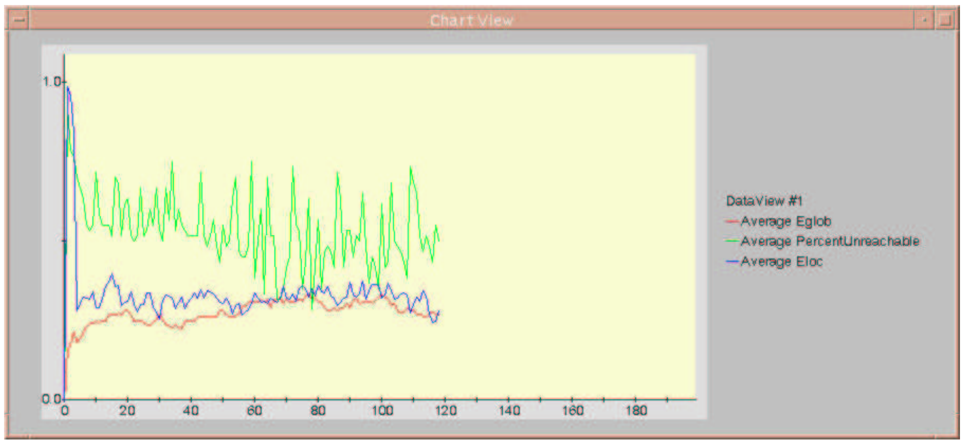

Figure 1: Metrics plotted versus time (rounds). Every peer has 5 friends and N=60. Keep-alive hop count=3. Friend list update mechanisms of original LOCKSS with a multicast group used

Figure 1 shows the performance of the original LOCKSS friend discovery algorithm where $N = 60$ with every new peer added directly to the friend list. In addition, there is a multicast group of 9 peers. Figure 2 shows our friend discovery

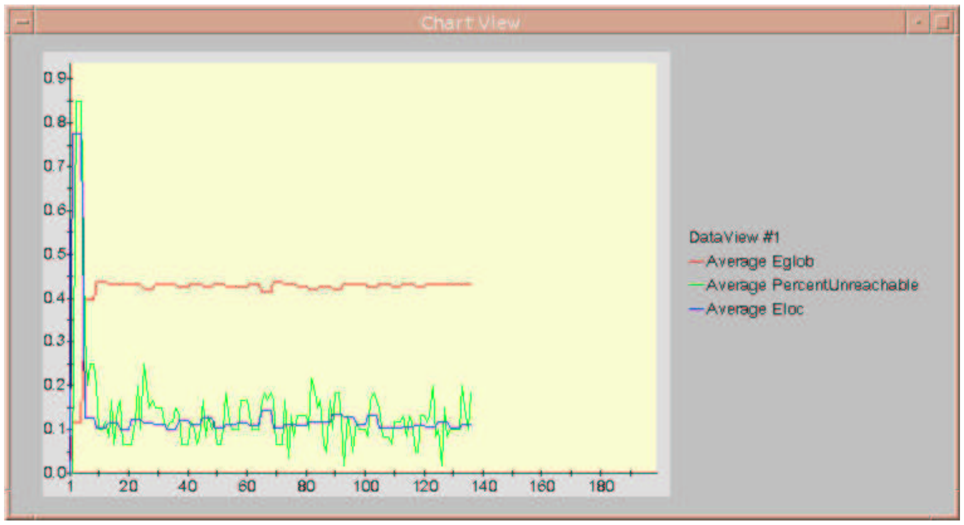

Figure 2: Metrics plotted versus time (rounds). Every peer has 5 friends and N=60. Keep-alive hop count=3. New friend list update algorithms used with a reference list without multicast

algorithm using the same N where the friend list is updated using the reference list. The performance of our algorithms is much higher than LOCKSS as observed by the high message reachability (low U_m) and the higher global efficiency. This performance is achieved without even using a multicast group to flood the message. The reason for the better performance is that peers perform better random sampling and they tend to form a larger connected component rather than smaller clusters. The fact that the local efficiency is lower in our algorithm verifies that fact.

In addition, we observed the global efficiency decreases as N increases, but not rapidly enough to concern us about the connectivity of the network. Figure 3 shows how the global efficiency decrease as N increases from 10 peers to 10000 peers. The simulation was run with 4, 8, 16 and 64 friends in the friend list. The more the friends the longer the global efficiency stays high.

Figure 4 and 5 show E_{glob}, E_{loc} and U_m for each round. For U_m we picked a random keep-alive message that was sent during that round. During these measurements every peer had 5 friends and $N = 100$. The difference between the two cases is that in Figure 4 no peer goes off-line, whereas in Figure 5 every peer can go off-line before each round with probability 0.4. Once peers go off-line the unreachability of each message is increased from about 0.25 to about 0.75. This means that messages propagate much more slowly when links are lost. We can remedy this by increasing the hop count of a keep-alive message. As shown in Figure 6, U_m decreased to about 0.5 as the hop count was increased from 3 to 4 hops.

The only effect that broken links have on the friend discovery protocol is to slow down friend discovery until the memory fills up with enough entries. If a peer does not receive a keep-alive message from a peer in its reference list for a long period of time[5] it can assume the peer is off the network.

[5]A period several orders of magnitude greater than the advertisement period

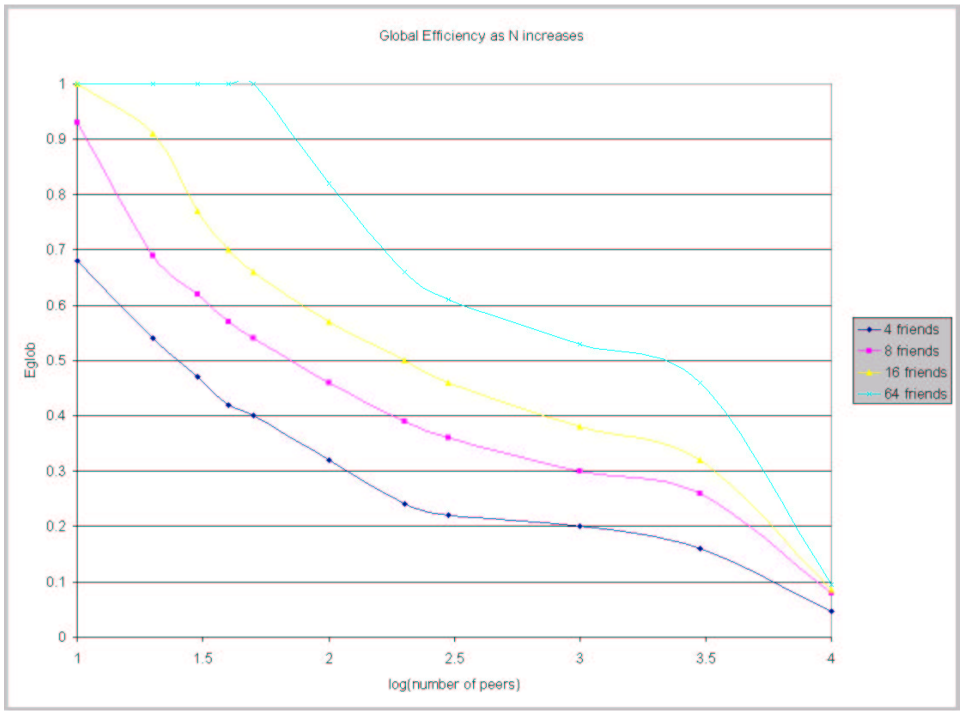

Figure 3: Global Efficiency versus log scale of the number of peers in the network. The four lines correspond to 4, 8, 16 and 64 friends in the friend list.

6 Opinion Polls

The Poll Protocol is used by a peer that wants to conduct an Opinion Poll. Such a peer is called the *initiator*. The peers that it calls to conduct the poll are called *participants*.

Before the initiator starts the poll it sets two values, the poll degree δ and the poll radius ρ. The poll degree sets the number of votes the initiator and each participant expects to count at the end. The poll radius defines the monitoring depth of the poll. A poll radius of 1 for example allows monitoring only for the initiator and a poll radius of 2 allows monitoring for the initiator and for the peers that monitor the initiator. Monitoring is described in more detailed in Section 6.0.3.

The brief description of an Opinion Poll is the following: the initiator floods a poll message with a certain degree and radius where every forwarder is considered a parent and the receivers are its children. Parent, grandparent and children form clusters together. Participants vote within their clusters and later they monitor all the votes within their cluster.

During an Opinion Poll the initiator and every participant maintains two lists: the *one-hop list* L and the *two-hop list* $L2$ regardless of the poll radius. The meaning of their names will become apparent after we describe the poll setup algorithm.

6.0.1 Poll Message

Poll messages contain the originator of a poll, the poll degree and the poll radius. In addition a poll message contains L as a set of peer IDs. Finally a hop count H that

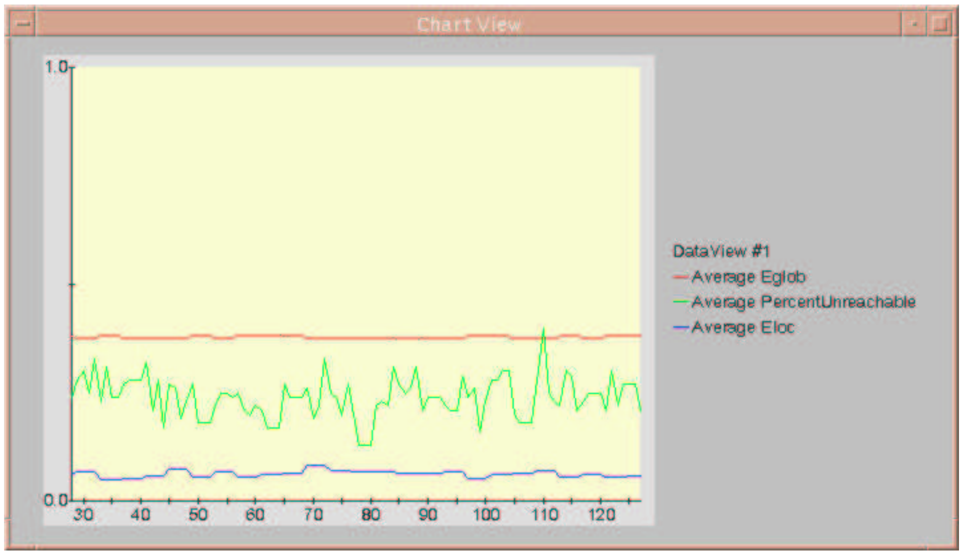

Figure 4: Metrics plotted versus time (rounds). Every peer has 5 friends and N=100. No peers fail. Keep-alive hop count=3.

is initialized at the value of the poll radius and is decremented by each subsequent forwarder.

6.0.2 Vote

A vote contains the voter and the sender of the vote. The voter attaches its response on the vote message, which could be a binary value (agreement or disagreement) or a piece of data to be verified by other peers.

6.0.3 Polling Protocol

An Opinion Poll consists of 3 phases: poll setup, voting and monitoring. The poll setup algorithm solves the problem of Poll Setup. The voting algorithm and the monitoring algorithm solve the Voting and Monitoring problem.

The poll setup algorithm is started by the initiator. It decides the poll size and the monitoring depth and sets ρ and δ accordingly. It randomly selects δ peers from its list (buffer or reference) and enters them in its one-hop list L. These peers are considered its children. It is up to the peer to decide whether it has enough peers in memory so that its selection of the one-hop list is unbiased. For example the peer might wait until it knows 100 other peers in order to pick 8 for the one-hop list, while another might think 20 is good enough.

The initiator creates a Poll message with a new ID and adds the following: (ρ, δ, $curHop$, L) where $curHop = \rho$ and is decremented by the receivers. The initiator sends the message to each peer in L. After the poll setup terminates the initiator becomes a voter.

Every receiver of the poll message performs the following:

- it becomes a voter for the poll-id of the message. This means that it will run the voting algorithm after poll setup terminates.

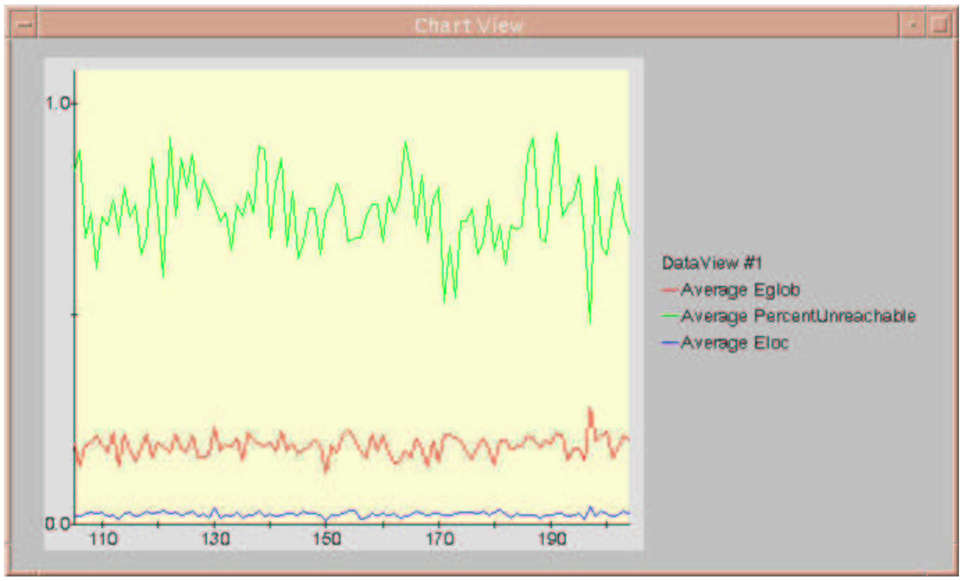

Figure 5: Metrics plotted versus time (rounds). Every peer has 5 friends and N=100. Peers can fail independently with 0.4 probability per round. Keep-alive hop count=3.

- it adds the entries of L, which is sent in the message to its own $L2$.
- it decrements $curHop$. If $curHop = 0$ it adds the sender to its L and stops. Otherwise it proceeds to the next step.
- it fills its own L with $\delta - 1$ randomly selected peers from its memory. Now L contains its children.
- it forwards L to the sender. The sender adds these entries to its $L2$.
- it adds the sender to L. Now L contains the children and the parent, a total of δ peers.
- in the poll message it replaces the sender's L with its L.
- it forwards the message to its L except the sender.

This algorithm terminates when curHop goes to 0.

After the algorithm terminates every peer's L contains peers that were one hop away in the flooding path of the poll message. $L2$ on the other hand contains peers that were indirectly introduced to each other through the contents of the poll message.

Examining the topology of the graph that this algorithm creates we observe the following:

- Direct links are formed from a peer to peers in its L and $L2$. Every link between two peers is bidirectional.
- The initiator and its immediate children in L all have links to each other and they form a clique if none of them goes off-line while the poll takes place. We call this the *main cluster*.

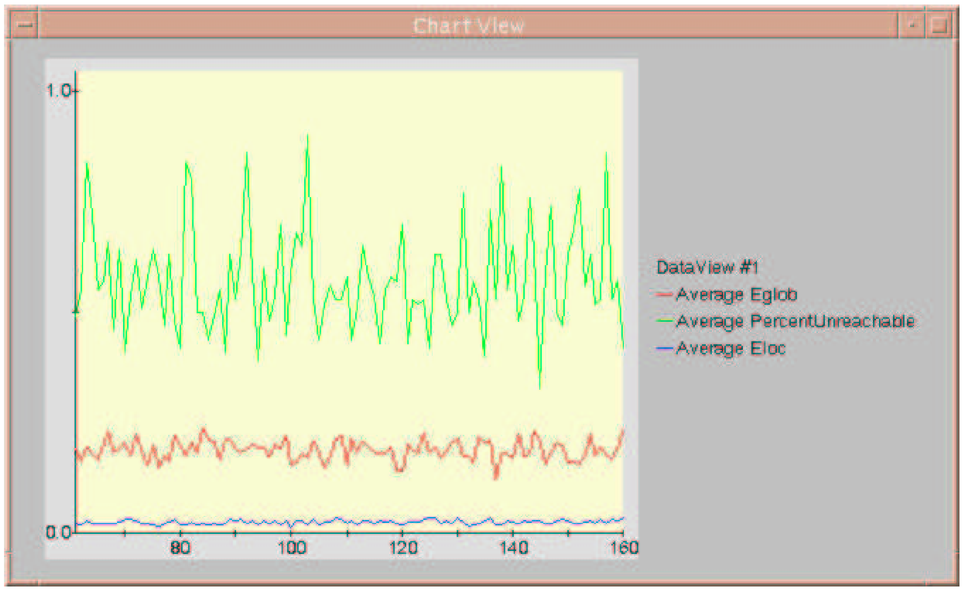

Figure 6: Metrics plotted versus time (rounds). Every peer has 5 friends and N=100. Peers can fail independently with 0.4 probability per round. Keep-alive hop count=4.

- Every peer $\rho - 1$ hops away from the initiator is a parent. The subgraph that contains a parent and the peers in its L is a clique as well if peers don't go off-line while the poll takes place. We call it a *secondary cluster*.

An example of how the poll setup phase works is shown in Figure 7 where the initiator is peer 0 and has set $\rho = 2$ and $\delta = 3$. In parentheses are the peer IDs the L entries. At the end of every hop the two-hop links are formed and the back links to the sender.

After poll setup is done every voter runs the voting algorithm. Every peer creates a vote (maybe after a small delay to wait for everyone to receive the poll message) and sends the vote to its L and L2 using a hop count of 2. Each peer that receives the vote forwards it to both L and L2. After a certain timeout every peer removes L and L2.

The initiator is the only peer that will have to update its data item based on the results of the poll. After it receives votes from the main cluster it groups the copies for each vote together. If no failures or malicious attacks have happened the initiator expects to receive $\delta - 1$ copies from each voter in the main cluster. For each voter it computes the *apparent* vote as the value of the majority of the copies. The final result is the value of the majority of the apparent values. As long as the majority of the peers in its one-hop list are not corrupted then it will receive the correct result. In case the majority function yields no winner ,then the initiator has two choices: if there is a vote that matches its own then it takes this as the majority value. If not, then it considers the vote as a negative vote.

The graph that is created by the poll setup phase enables monitoring votes as well. Otherwise, there would be no reason to create the extra links among peers. We could instead have the poll initiator set $\rho = 1$ and have only the its children vote and send the votes directly to the initiator using a star topology with the initiator at

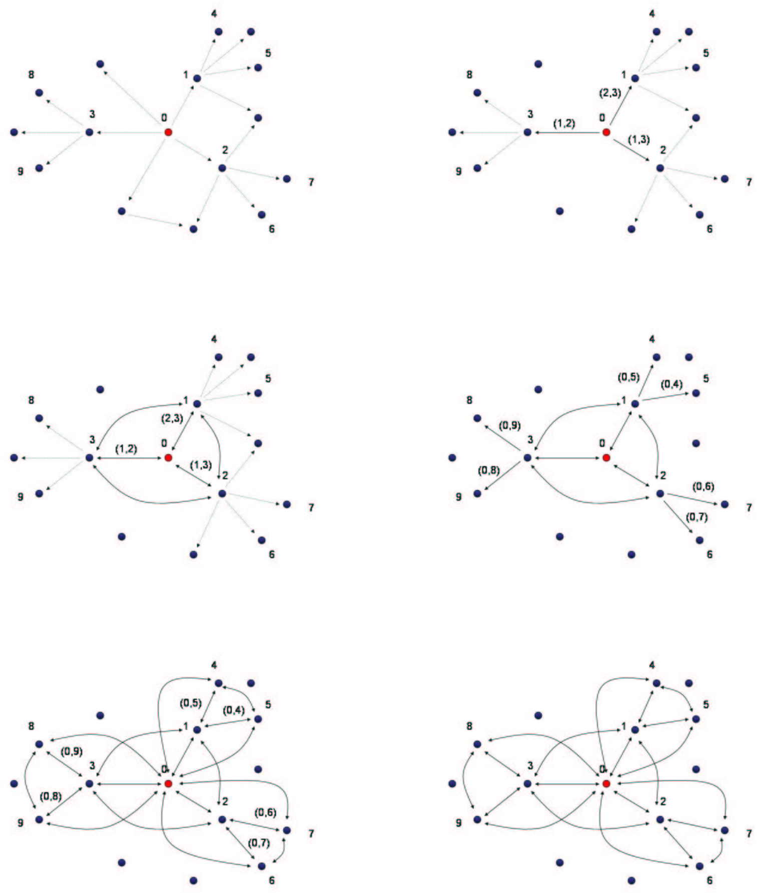

Figure 7: An example of the setup phase of a poll with $\delta = 3$ and $\rho = 2$. Six steps are shown from top left to bottom right. At the end there are 4 monitoring clusters formed.

the center.

The monitoring algorithm works as follows: every peer monitors votes originated by peers in a cluster that it belongs to. It records the votes that it sees, identifies the originator and the forwarder of the votes it is monitoring and it groups the copies of the same vote together. Finally it computes for each voter the *apparent* vote value as the value of the majority of the copies for this particular voter. Therefore every monitoring peer has a view of the votes cast by peers within its own cluster.

The larger the poll radius the more peers get monitored. However, $\rho = 2$ allows deep enough monitoring for a single poll. Also a good idea is to set δ to an odd number value so there are no ties estimating the apparent vote for each voter. There might be a case when there is no majority found when estimating the apparent vote. In that case the monitoring peer acts as the initiator does when estimating the apparent vote. In addition it records all the false values it received if none of them matches its own vote.

Every peer unsatisfied by the view that it has after monitoring is allowed to perform subsequent polls. There are two cases that a peer might be unsatisfied by the result. The initiator is considered unsatisfied if the results of the voting are very close, so it can call another poll on a larger peer set. A participant is considered unsatisfied if the majority of the votes it received disagrees with its vote. That peer should initiate a poll of its own to possibly a larger set of peers.

An unsatisfied monitoring peer that performs its own poll can evaluate the integrity of the previous poll and possibly detect malicious and corrupted peers as described in more detail in Section 7.

6.1 Evaluation

Unlike LOCKSS we used two different lists L and $L2$ to perform message routing in polls instead of using the original friend list. The reason for doing that was that a random graph where each peer has a constant number of links has very low "cliquishness"; in other words low local efficiency. From our simulation we observed that the local efficiency decreases much more rapidly than the global efficiency as the size of the network increases. Figure 8 shows how the local efficiency decreases as N increases from 10 peers to 10000 peers. The simulation was run with 4, 8, 16 and 64 friends in the friend list. The local efficiency drops significantly as N increases even if the number of friends increases, which means that we cannot depend only on the immediate friends to form a good cluster for doing a poll.

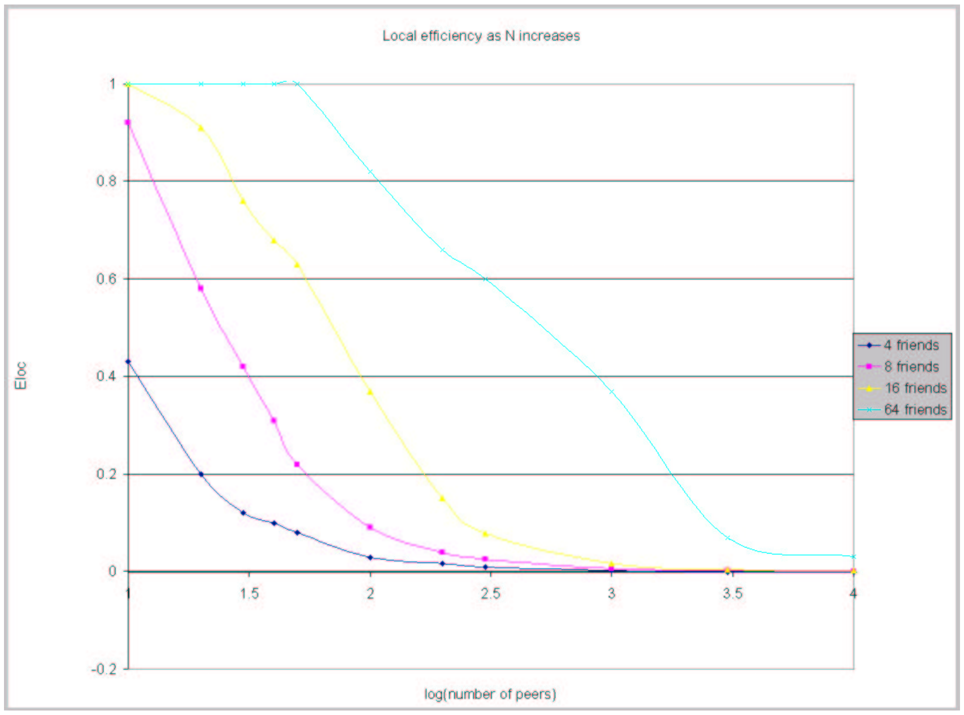

Figure 8: Local Efficiency VS log scale of the number of peers in the network. The four lines correspond to 4, 8, 16 and 64 friends in the friend list.

The results follow our intuition from random graphs that the graph tends to form a large connected component rather than a group of clusters when each node has a constant number of links greater than 1. These results led us to form the temporary links for the poll protocol to increase significantly the local efficiency of the poll network.

That lead us to have our Polling Protocol go through the Poll Setup phase to increase the local connectivity of the poll participants. Table 1 shows some values that we collected from our simulation with $N = 100$ and $\rho = 2$ as δ increases. In general the local connectivity stays above 0.9.

δ	E_{loc}
3	0.90 - 0.91
4	0.92 - 0.93
5	0.92 - 0.93
9	0.90 - 0.91

Table 1: Local efficiency of a poll for different values of $delta$. $N = 100$ and $\rho = 2$

When peers go off-line the two things we looked at was the actual number of participants in the poll and secondly, the number of copies of each vote that every participant received at the end of the poll. These two values were compared with the ideal case where all peers all on-line at all times.

peer_id	number of vote copies
0	1
34	5
5	5
2	5
99	3
97	3
81	3
68	3
45	3
30	3

Table 2: Number of copies that peer 0, the initiator of the poll received. 9 other peers participated. No peer went off-line during the poll

peer_id	number of vote copies
0	1
99	3
71	2
8	2

Table 3: Number of copies that peer 0, the initiator of the poll received. 3 out of the 9 expected participants. Participants went off-line mostly in the setup phase.

Tables 2, 3 and 4 show examples of 3 different cases of poll outcomes for the poll initiator. The first case is when all peers are on-line at all times. The other two cases show the results when every peer can go off-line with probability 0.4 before each round. A poll is simulated as 2 rounds. The first round corresponds to

peer_id	number of vote copies
0	1
85	3
80	3
47	2
42	3

Table 4: Number of copies that peer(0), the initiator of the poll received. 6 out of 9 participants. Participants went off-line mostly in the voting phase.

the setup phase where peers create the temporary links to other peers. The second round corresponds to the voting phase, where peers cast their votes to all the peers they know. Peers could fail before either of the two phases. Table 3 shows the results when most of the failures happen before the poll setup phase. As you can see the number of participants is much smaller than it should be because many peers failed before they were called to participate. Table 4 shows the results when most of the failures happen during the voting phase. There are more peers to vote, but the number of copies is lower than the ideal case.

These results show that although we have a very high off-line rate, the polls still work well and the poll initiator receives a handful of copies. The numbers are similar for the rest of the participants in the poll.

7 Attacks and Defenses

There are three types of attacks an adversary can perform. First, impersonation attacks, second attacks on the Friend Discovery Protocol, and third on the Poll Protocol.

We divide attacks on the protocols into the following classes:

- Communication attacks that abuse the communication constraints such as exceeding the number of keep-alive messages per time period, exceeding the number of entries in the friend list or not forwarding messages while the hop count is not 0.
- Timing attacks where the adversary tries to gain advantage over other peers by sending messages at specific times.
- Abuse of probability attacks where the adversary will seek a deterministic behavior whenever a loyal peer might act randomly.
- Corruption attacks. These involve all the cases that the adversary will try to provide false information to other peers or alter message contents such as vote values.

7.1 Impersonation Attacks

As stated in our assumptions a peer can send any message to whoever it wishes, however it can only read messages that are addressed to its own IP. Therefore, in our system the IP address of the peer is a form of weak authentication, where every peer can verify messages sent by a peer using well known methods of 3-way handshakes or 1-way handshakes (*find references to that*). Therefore we assume that such message can only do as much damage as denial of service. But in a peer to peer system with exponential network topology denial of service attacks do not have as strong an effect as shown in [11] and [12].

7.2 Friend Discovery Protocol attacks

An adversary can perform timing and communication attacks in order to affect the contents of the friend list of the targeted peer. Since the friend list is selected periodically from memory the only way a malicious peer can enter itself in the friend list is if it gets itself inserted in the peer's memory. Therefore a malicious peer will try to send keep-alive messages to as many peers as possible and we can assume that it can reach every peer on the network.

It is very difficult to detect such an abuse of communication, since it requires aggregate data from many peers. The best we can do is to limit the effects of such an attack by combining memory size and listening behavior.

There are two cases depending on the memory model of the attacked peer:

In the case of a peer that keeps a reference list the attack is very weak since we assume that the reference list approaches the properties of the infinite list. Every peer in the network has probability near $\frac{1}{number\_of\_peers}$ to be selected as a friend so since we assume that the majority of the peers are loyal then the friend list will contain more loyal peers than malicious.

In the case of a peer that keeps a buffer list the attack can be very strong if the behavior of the attacked peer can be predicted. If the adversary knows that the peer listens at all incoming messages and updates its friend list at the end of a certain time period with the peers that it remembers more recently then the adversary will synchronize its keep-alive message to be sent right before the peer decides to update its friend list from its memory. Therefore the listening behavior of a peer with limited memory should be unpredictable.

As stated earlier we concern ourselves with the former case. A peer with a reference list can easily detect if a peer sends messages too frequently by doing statistical analysis on the number of messages received by each peer in its list.

The worst case that we have observed is when a peer just enters the network. If the peer enters the network not knowing a single loyal peer then it will never learn any and live in a network full of malicious peers. Then it can easily be subverted to carry a corrupted data item. Even if that worst case scenario happens, the subverted peer would still be loyal, but corrupted. Since this peer will get isolated by any good peers so its corrupted data items cannot harm other peers. As long as a large proportion of the nodes is loyal and healthy if this node obtains a single link to a loyal peer it will eventually learn the rest. If it tries to spread corrupted content by participating in Opinion Polls it will become healthy again.

7.3 Poll protocol attacks

A malicious peer can tamper with the poll protocol in either of two roles: as a poll initiator or as a participant.

As a poll initiator it can perform a communication attack and *fix* an Opinion Poll in order to subvert loyal and healthy peers. This is done by fixing the participants to be all its malicious friends and a single healthy peer. To the attacked peer the poll looks like a normal Opinion Poll, but its L and $L2$ lists contain malicious peers, so the clusters it belongs to will yield corrupted results.

However this attack has little effect if the healthy participant has connections to loyal peers and it can pick its own children (i.e. it is less than ρ hops away from the initiator). Even in the case where the participant is ρ hops away, the peer enters an unsatisfied state and then it calls a its own Opinion Poll of larger size than the previous poll and then detect the attack.

As a participant the malicious node can only have an effect on clusters it belongs to. It can perform attacks at the poll setup phase and the voting phase.

During the poll setup phase it can only have an effect on its children. Every loyal peer following the protocol forwards the poll message downstream to a fixed number δ of peers. So every receiver expects to hear only from one parent. If it hears from more then it means it's either a child of both or one is a malicious peer trying to corrupt it. For this to happen the malicious peer has to know what are the children of the other peers, which is highly unlikely if they are chosen randomly from memory. Even if it succeeds this has the same effect as acting as an initiator on a secondary cluster.

During the voting phase a malicious participant can perform two types of attacks. The first involves sending votes with a false value and possibly changing good votes that it receives to the same false value before forwarding. The purpose of this attack is to convince the poll initiator and the loyal monitoring peers to accept the false value as correct. The second type of attack involves either dropping or modifying votes such that they don't match the correct value, but not in a consistent way. The purpose of this attack is to corrupt votes from loyal peers, such that the initiator or a monitoring peer cannot determine what value an apparent vote has.

The first type of attack can be easily detected by the monitoring peers. They record the votes they receive and they notice apparent votes that disagree with their vote. Because they are unsatisfied they call their own polls to verify the votes that they received earlier. By comparing the new values with the old values they can detect the attack.

The initiator will not get corrupted by such an attack as long as more than half of its children in the one-hop list are loyal.

The second type of attack is very similar to the Byzantine general's problem [6], where a voter acts as the commanding general and the peers in its cluster as the lieutenants. A peer in the cluster cannot determine whether the voter is corrupted or the forwarder has corrupted a perfectly good vote. Since these two cases are indistinguishable to a third peer, peers keep a neutral position regarding the inconsistency.

8 System Steady State

In this section we analyze the behavior of the system in the long term. In particular we calculate the probability that the system will fail to preserve a data item given the number of loyal but corrupted peers i and the number of malicious peers m. We let N_l be the number of loyal peers so the total number of peers is $N_l + m$ and the number of loyal and healthy peers is $N_l - i$. We assume every poll has cluster size of n. From our attack and defense analysis we can assume that a malicious peer participating in a poll can perform at most as much damage as a loyal corrupted peer.

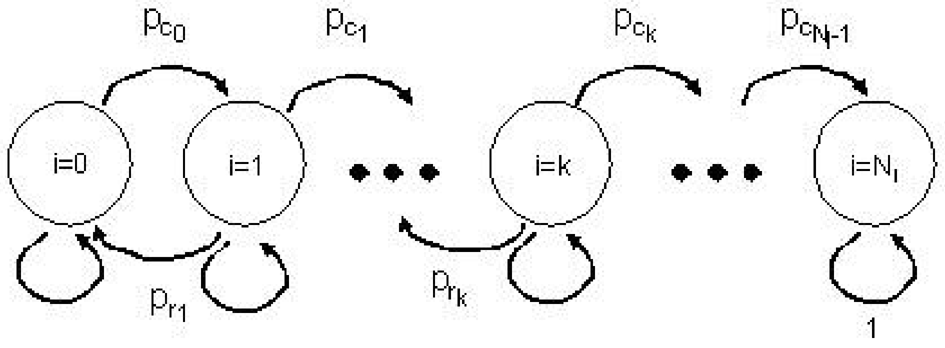

Figure 9: Markov chain diagram of the system.

We model the system as a Markov chain of length $N_l + 1$ where each state represents the system when i corrupted peers exist, as shown in Figure 9. The last state, N_l is an absorbing state and state 0 is an absorbing state if $m \leq \frac{n}{2}$. The system is in a state i either because of malicious attacks or because of natural data corruption. A transition to a new state happens when a loyal peer conducts a poll.[6] Given a state i the system can move to $i+1$ if a poll corrupted the initiator (with probability p_{c_i}). The system goes to state $i-1$ if the poll repaired a corrupted initiator (with probability p_{r_i}). If the initiator did not change from corrupted to healthy or vice versa then the system stays at i. To see whether the system fails or not in the long term we are interested in calculating the probability P_i, which is the probability that given the system starts at i it will reach $i = N_l$.

Intuitively, we expect the system to have two stable regimes, one where no corrupted copies survive (state 0) and one where no healthy copies survive (state N_l). Any other state should be unstable and in the long run will end up in one of the stable regimes.

8.1 Statistical Analysis

We define as p_H the probability that we select a healthy peer out of $N_l + m$ given i corrupted peers. So, $p_H = \frac{N_l - i}{N_l + m}$. We define P_{poll} as the probability that a poll of cluster size n will agree in favor of the healthy copy, in other words that the majority of the voters are healthy and loyal. We assume that $N >> n$, so

$$P_{poll} = \sum_{r=\lceil \frac{n}{2} \rceil}^{n} \binom{n}{r} p_H^r (1 - p_H)^{n-r}$$

This is the same probability that a r-out-of-n component system will survive if more than half of its components remain functional.

Therefore,

$$p_{c_i} = p_H(1 - P_{poll})$$

[6]We assume polls happen more frequently than natural corruptions, so we don't consider them explicitly for the state transitions.

and

$$p_{r_i} = (1 - p_H)P_{poll}$$

Since our model has the Markov property we can compute P_i by conditioning on the outcome of the first poll:

$$P_i = p_{c_i}P_{i+1} + p_{r_i}Pi - 1 + (1 - p_{c_i} - p_{r_i})P_i \Rightarrow P_{i+1} - P_i = \frac{p_{r_i}}{p_{c_i}}(P_i - P_{i-1})$$

for $0 < i < N_l$.

Since $i = N_l$ is an absorbing state we get one additional equation,

$$P_{N_l} = 1$$

Solving this system of N_l equations we find P_i:

$$P_1 = \frac{1 + \sum_{k=1}^{N_l-1} \prod_{j=1}^{k} \frac{p_{r_j}}{p_{c_j}} P_0}{1 + \sum_{k=1}^{N_l-1} \prod_{j=1}^{k} \frac{p_{r_j}}{p_{c_j}}}$$

$$P_{i+1} = P_1 + \frac{\sum_{k=1}^{i} \prod_{j=1}^{k} \frac{p_{r_j}}{p_{c_j}} (1 - P_0)}{1 + \sum_{k=1}^{N_l-1} \prod_{j=1}^{k} \frac{p_{r_j}}{p_{c_j}}}$$

for $0 < i < N_l$.

In order to compute P_0 we use recursive approximation instead of computing P_0 explicitly. We start with $P_0 = 0$ and compute P_1. To find the new P_0 we set $P_0 = P_1 p_{c_0}$ since $p_{r_0} = 0$.

Figure 10 shows how P_i changes with respect to i. We can notice that even when $i = 20$ the system tends to go to the $i = 0$ regime almost certainly (with probability 0.9999), while when $i > 50$ the system tends to go to the $i = N_l$ regime. We see that as the number of malicious peers increases from $m = 10$ to $m = 30$ then the system can afford a smaller number of corrupted peers in order not to fail ($i < 10$). However, as the size of the poll increases to $n = 9$, the system can afford more corrupted peers ($i < 20$).

In Figure 11 we increase N_l to 500 and explore how the effect of the malicious peers is modified. We observe that although the ratio of malicious peers is the same and the size of the poll remains unchanged having more peers in total destroys the effect of malicious peers completely. For example, when the ratio is $\frac{30}{100} = \frac{150}{500}$ for $N_l = 100$ we get $P_{10} = 10^{-3}$ and for $N_l = 500$ we get $P_{50} = 10^{-13}$

The system shows very high resilience to intentional corruption, even if the number of malicious and corrupted peers is a high percentage of the total population. We assume that unintentional corruption is less dangerous than intentional, because it happens by natural failures that occur less often than attacks. Since the system tends to move to the $i = 0$ regime when there are is not a sufficient number of corrupted copies, then assuming that natural failures occur independently for each peer i will not deviate much from 0, so the system will recover easily.

9 Conclusions

We have designed and analyzed a set of protocols that achieve data resilience for the long term using a set of mutually untrusted peers that are loosely organized.

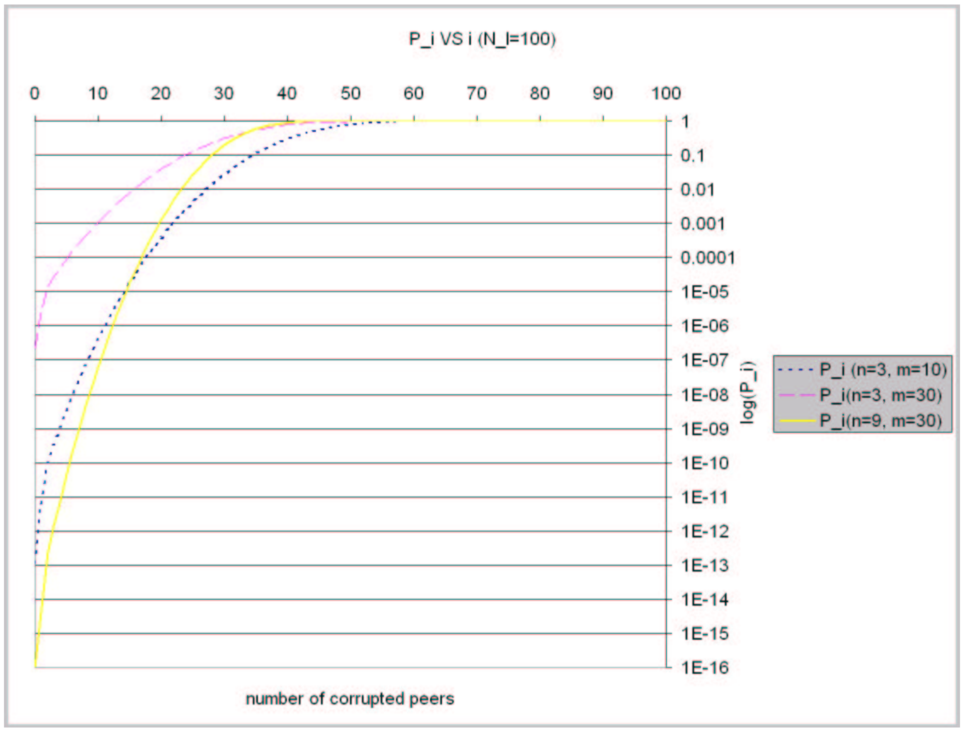

Figure 10: log scale of P_i VS i, the number of corrupted peers in the network. $N_l = 100$. The 3 lines correspond to (n=3,m=10),(n=3,m=30) and (n=9,m=30)

We use Opinion Polls to heal corrupted copies of data items stored by peers. Our algorithm scale to a large number of peers and can resist attacks by malicious peers and peer failures. We have evaluated our protocols with respect to existing methods for data preservation and shown that they can achieve similar effects with less expensive algorithms than traditional fault tolerance techniques. Finally our simulations show that they scale well as the number of peers increases.

10 Future Work

As we stated earlier peers with a buffer list can easily become targets for malicious peers, since malicious peers can perform timing attacks and get themselves in every peer's buffer list. A future project would be to find a good policy for replacing entries in the buffer list such that malicious peers cannot predict what messages a peer listens to and what entries it replaces in the buffer list.

11 Acknowledgments

We would like to thank Steve Heller, Mark Moir and Victor Luchangco for giving us feedback on the design of our protocols and their evaluation. Also Mark Seiden for discussions that were instrumental in setting the goals and deriving the design for the initial LOCKSS algorithms.

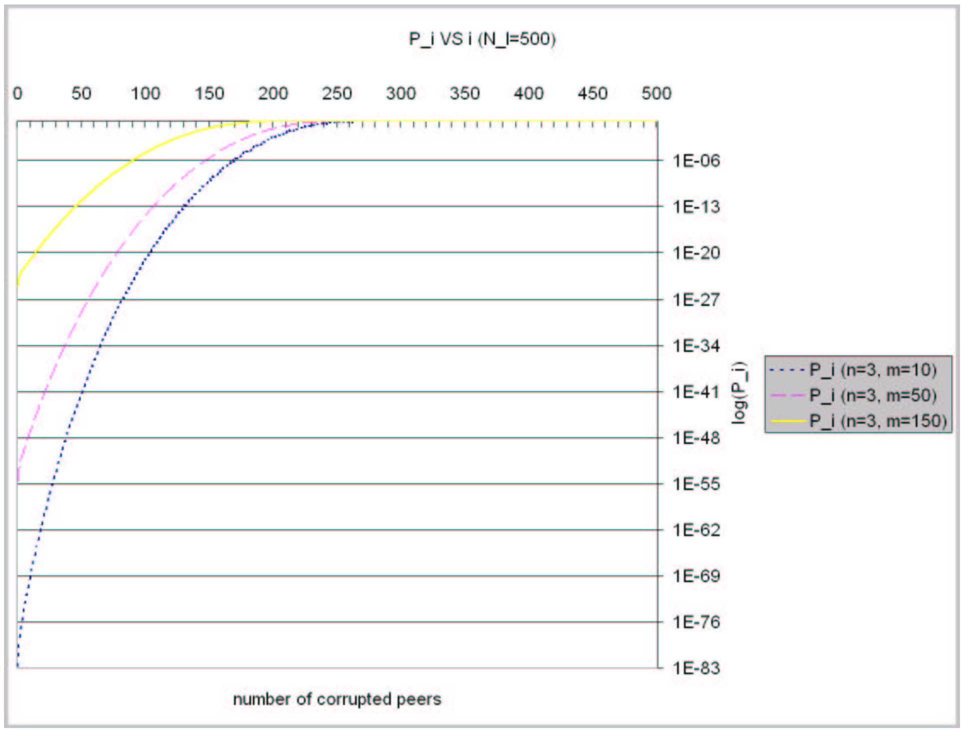

Figure 11: log scale of P_i VS i, the number of corrupted peers in the network. $N_l = 500$. The 3 lines correspond to (n=3,m=10),(n=3,m=50) and (n=3,m=150)

References

[1] D. Rosenthal and V. Reich, "Permanent web publishing," *Freenix*, June 2000.

[2] P. Maniatis and M. Baker, "Enabling the archival storage of signed documents," *USENIX Conference on File and Storage Technologies*, January 2002.

[3] "Gnutella," http://www.gnutella.com.

[4] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," *Lecture Notes in Computer Science*, 2001.

[5] B.F. Cooper and H. Garcia-Molina, "Peer-to-peer trading to preserve information," *ACM TOIS*, 2002.

[6] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, 1982.

[7] B. Lampson, "How to build a highly available system using consensus," *Distributed Algorithms - Lecture Notes in Computer Science*, pp. 1–17, 1996.

[8] M. Castro and B. Liskov, "Practical byzantine fault tolerance," *Third Symposium on Operating Systems Design and Implementation*, 1999.

[9] E. Sit and R. Morris, "Security considerations for peer-to-peer distributed hash tables," *1st International Workshop on Peer-to-Peer Systems*, 2002.

[10] S. Floyd et al, "A reliable multicast framework for light-weight sessions and application level framing," *IEEE/ACM Transactions on Networking, Vol 5, Num 6*, 1997.

[11] R. Albert, H. Jeong, and A. Barabasi, "The intenet's achilles' heel; error and attack tolerance of complex networks," *Nature*, 2000.

[12] P. Keyani, B. Larson, and M. Senthil, "Peer pressure: Distributed recovery from attacks in peer-to-peer systems," *International Workshop on Peer-to-Peer Computing*, 2002.

[13] M. Parker, "Ascape: an agent-based modeling framework in java," *http://www.brook.edu/dybdocroot/es/dynamics/models/ascape/*, 1999.

[14] V. Latora and M. Marchiori, "Efficient behavior of small-world networks," *Phys. Rev. Lett. 87*, 2001.

[15] "The mica wireless measurement system," http://www.xbow.com/Products/Wireless_Sensor_Networks.htm.

About the Authors

Nikolaos Michalakis

N. Michalakis is with the Massachusetts Institute of Technology. This work was done while an intern at Sun Microsystems Laboratories.

Dah-Ming Chu
Sun Microsystems Laboratories

Dah Ming Chiu is a research at Sun Microsystems Laboratories in Burlington, Massachusetts. His recent research is on reliable multicast protocols and multicast flow congestion control. Prior to Sun, he worked at Digital Equipment Corporation, and AT&T Bell Labs. His research interests include performance modeling and analysis of network protocols, distributed systems, and WWW-based applications. He received a Ph.D. Degree from Harvard University and a B.Sc. Degree from Imperial College, University of London.

Dr. David S. H. Rosenthal
Formerly, Distinguished Engineer
Sun Microsystems Laboratories

Dr. David Rosenthal is investigating techniques for distributed fault tolerance in a project jointly funded by Sun Labs, the Andrew W. Mellon Foundation, the National Science Foundation and Stanford University Libraries. The project is aimed at long-term preservation of the web editions of academic journals, such as those published by Stanford's Highwire Press.

David joined Sun in 1985 from the Andrew project at Carnegie-Mellon University. He worked on window systems and was part of the team which developed the X Window System, now the open-source standard. He also worked on graphics hardware, the operating system kernel, and on system and network administration.

David left Sun in 1993 to be Chief Scientist and employee #4 at Nvidia, now the leading supplier of high-performance graphics chips for the PC industry. He worked on I/O architecture. In 1996 he joined Vitria Technology, now a leading supplier of e-business infrastructure technology. He worked on reliable multicast protocols and on testing industrial-strength software. In 1999 he re-joined Sun Microsystems Laboratories as a Distinguished Engineer for a brief period before moving on to the LOCKSS project at Stanford.

David received an MA degree from Trinity College, Cambridge and a Ph.D. from Imperial College, London. He is the author of several technical publications and holds 23 patents. His interests include backpacking and the theatre.