Topology Sensitive Replica Selection

Dmitry Brodsky, Michael J. Feeley, and Norman C. Hutchinson
Department of Computer Science
University of British Columbia
{dima,feeley,norm}@cs.ubc.ca

Abstract

With the proliferation of peer-to-peer storage it is now possible to protect data at a level that is comparable to traditional replication systems but at reduced cost and complexity.

These peer-to-peer storage systems must be able to replicate data on hosts that are trusted, secure, and available. However, recent research has shown that the traditional model, where nodes are assumed to have identical levels of trust, to behave independently, and to have similar failure modes, is over simplified. Thus, there is a need for a mechanism that automatically and efficiently selects replica nodes from a large number of available hosts with varying capabilities and trust levels.

In this paper we present an algorithm to handle replica node selection either for new replica groups or to replace failed replicas in a peer-to-peer storage system. We show through simulation that our algorithm maintains the node inter-connection topology minimizing the cost of recovery from a failed replica, measured by the number of nodes affected by the failure and the number of inter-node messages.

1 Introduction

Over the last few years the popularity of peer-to-peer storage systems has increased at a phenomenal rate. Many systems [6, 9, 10, 15, 16] were created primarily to share information, while other systems [2, 5, 11, 12, 18] were designed to function as deep archival repositories.

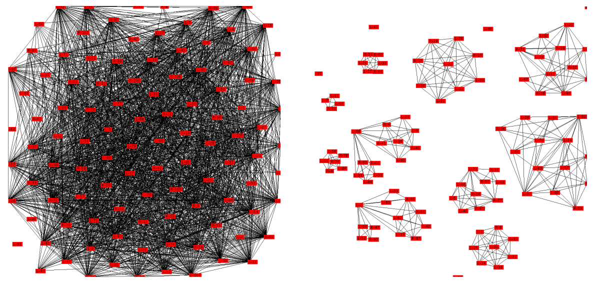
In general, with today's peer-to-peer storage it is possible to protect data at a level that is comparable to traditional replication systems but at reduced cost and complexity. Peer-to-peer storage systems provide the needed flexibility, reliability, and scalability to operate in present day environments, and handle present day loads.

Most recent peer-to-peer systems [7, 19, 20] select replica nodes using a quasi-random process. Systems that randomly select nodes assume that the cost of replication and the level of reliability are uniform across the system. In addition, they assume that the failure modes of the nodes are independent. Although, these assumptions may be true in the ideal world and are reasonable for simulation purposes, they are generally over simplified [13, 24].

To overcome the limitations of random node selection, several systems [1, 4, 8, 21] take a more systematic approach, selecting replica nodes based on node characteristics such as replication costs, reliability levels, location, and bandwidth. These systems are thus able to effectively place replicas to maximize the data's safety while minimizing the overall cost of replication.

We believe that node selection should not only depend on node characteristics, but also on the topology of the system, because a system's topology can greatly influence operation. The effect of poor replica node selection is twofold. First, the safety of data is jeopardized by placing it on poorly suited nodes. Second, a large number of inter-node connections are created which can greatly increase the cost of recovery. These inter-connections, physical [8] or virtual [4], are created when a set of replica nodes cooperate to store an object. Each object may potentially create new inter-node connections. Thus, as the system evolves (nodes fail and are replaced) and objects are created, every replica node may become connected to every other replica node, making group communication and recovery costly.

In this paper we present an algorithm, called *TopSen*, that performs replica node selection for a peer-to-peer storage system. *TopSen* selects replica nodes for data based on the topology of the system and node characteristics. The algorithm selects replica nodes that satisfy user specified constraints, such that the number of new interconnections between the nodes is minimized. This minimizes the potential increase in communication necessary to handle a failure. The driving goal of *TopSen* is that the number of nodes affected by a failure of a node should be minimized.



(a) Random (b) TopSen

Figure 1: Topology after ten years of simulation of 100 nodes using Random and TopSen replica node selection algorithms.

2 Overview

A peer-to-peer storage system consists of a collection of nodes (physical machines) that store data (files or blocks). Data is replicated or erasure encoded on a number of hosts to ensure its availability and reliability in the event of node failure. When a new object is created, the system selects a set of nodes to store the object. When a sufficient number of nodes have failed, the system selects replacement replica nodes for the objects on the failed ones, and re-replicates the data.

The number of inter-connections between a node and the other nodes directly impacts the number of nodes affected when the node fails; a large number of inter-connections implies a large number of nodes involved in the recovery. If node selection is performed haphazardly then eventually a single node failure could affect a very large number of nodes.

DHT based peer-to-peer storage systems [7, 20] implicitly minimize the number of inter-connections by explicitly specifying the replica nodes a node replicates to based on node IDs. Unfortunately, this same mechanism does not allow these systems to select replica nodes based on node characteristics; thus, they must assume that node reliability and failure modes are uniform across the entire system. Removing this replica node selection constraint results in a system, similar to [1], that more or less randomly selects replica nodes.

In a peer-to-peer storage system where replica nodes are selected at random (Figure 1a) the topology after ten years of simulation of 100 nodes is such that nodes are highly inter-connected. Thus, a single node failure affects a large number of nodes. In contrast, our approach, shown in Figure 1b, minimizes the inter-node connectivity such that a replica node failure affects only a small number of nodes.

Thus, the first goal of TopSen is to minimize the number of inter-connections among replica nodes so that the result is a sparse forest of small clusters (components). The second goal of TopSen is to balance the load. Otherwise, the first goal can easily be achieved by storing all data on the same set of nodes.

We achieve the above goals in three ways. First, when selecting replica nodes to replicate a new file or to replace a failed replica node TopSen attempts to use nodes that are not connected to any other nodes (*singletons*). The selected nodes thus form a small disjoint component, underutilized nodes are used, and new inter-connections to existing components are not created.

Second, if no singletons are available then an existing set of nodes is reused if they satisfy the replication criteria. Although, this imposes additional load on that set of nodes, it does not introduce any new inter-connections.

Third, TopSen splits apart components when replacing a failed replica node and during periodic topology maintenance. Components are sometimes joined to satisfy replication requirements, in many instances though, a single node x joins the two or more mostly independent components. The topology can be improved by splitting these weakly connected components. Thus, if node x fails and there are singleton nodes available, then TopSen replaces x with two or more singletons. In addition, the topology is periodically examined and weakly connected components are split. By combining these three methods the algorithm maintains a better topology than other simpler approaches as shown in Section 4.

3 Algorithm

In this section we describe the TopSen algorithm, which is presented as three separate algorithms. The first algorithm, presented in Section 3.2, is used to select the set of nodes to store a newly created file. The second algorithm, presented in Section 3.3, is used to select a replacement node for one that has failed. The third algorithm, presented in Section 3.4, is used to perform the periodic maintenance on the topology.

3.1 Definitions

First we define the terms used to describe the algorithms, which are presented as algorithms on graphs. We use the term *vertex* to denote a node. Edges (undirected) represent inter-connections between nodes. The term *topology* $\mathcal{T}$ refers to the graph made by the vertices and the edges; $|\mathcal{T}|$ refers to the number of nodes in the system.

The term *singleton* is used to denote a node that is storing zero replicas and is not connected to any other node. We use the term *cut vertex* to describe a node that if removed disconnects the component into two or more components.

A node, or a set of nodes, is selected based on a *node selection constraint* which is defined as a set $\mathbb{K}$ of pairs consisting of an *ID*, and a constraint function $\Phi()$. Given a value v the function $\Phi()$ returns true if v satisfies the constraint.

A node's type or characteristic is defined as a set $\mathcal{C}$ of pairs, $\langle ID, v\rangle$, consisting of an *ID* and a value v for each characteristic. Given a node x and a node selection constraint $\mathbb{K}$ we say that x *satisfies* $\mathbb{K}$, denoted $\mathcal{C}_\mathsf{x} \subseteq \mathbb{K}$, if for all $k \in \mathbb{K}$ there exists $c \in \mathcal{C}_\mathsf{x}$ such that $ID_c = ID_k$ and $\Phi_k(v_c)$ is true. Similarly, $\mathbb{K}$ satisfies $\mathbb{K}'$, $\mathbb{K} \subseteq \mathbb{K}'$, if for all $\mathcal{C}$, if $\mathcal{C} \subseteq \mathbb{K}$ then $\mathcal{C} \subseteq \mathbb{K}'$.

The term *policy* is used to denote the replication requirements for a file or a set of files. A policy $\mathcal{P}$ consists of a set N of nodes, a *policy specifier* $\mathbb{L}$, and a *load* property. The *policy specifier* $\mathbb{L}$ specifies the composition of $\mathcal{P}$ and is defined as a set of pairs, $\langle \mathbb{K}, k\rangle$, consisting of a node selection constraint $\mathbb{K}$ and a non-negative integer k for each type of node required. The *load* specifies the number of times the policy has been reused, or the number of files that use the policy. A set of nodes N satisfies policy specifier $\mathbb{L}$ if there is a function that maps the nodes in N to the tuples in $\mathbb{L}$, such that for each $\langle \mathbb{K}, k\rangle \in \mathbb{L}$ at least k nodes satisfying $\mathbb{K}$ are mapped to it. A policy specifier $\mathbb{L}$ satisfies policy specifier $\mathbb{L}'$, denoted $\mathbb{L} \subseteq \mathbb{L}'$, if every set of nodes that satisfies $\mathbb{L}$ also satisfies $\mathbb{L}'$. A policy $\mathcal{P}$ satisfies policy $\mathcal{P}'$, denoted $\mathcal{P} \subseteq \mathcal{P}'$, if $\mathbb{L}_\mathcal{P} \subseteq \mathbb{L}_{\mathcal{P}'}$

Two policies specifiers $\mathbb{L}$ and $\mathbb{L}'$ can be combined, denoted $\mathbb{L} + \mathbb{L}'$, to create a new policy specifier $\mathbb{L}''$. $\mathbb{K}_{\mathbb{L}''} = \mathbb{K}_\mathbb{L} \bigcup \mathbb{K}_{\mathbb{L}'}$. Let $l \in \mathbb{L}$, $l' \in \mathbb{L}'$, and $l'' \in \mathbb{L}'$, if $\mathbb{K}_l$ and $\mathbb{K}_{l'}$ are the same then $k_{l''} = k_l + k_{l'}$.

Two policies $\mathcal{P}$ and $\mathcal{P}'$ can be combined, denoted $\mathcal{P} + \mathcal{P}'$, to create a new policy $\mathcal{P}''$. $N_{\mathcal{P}''} = N_\mathcal{P} \bigcup N_{\mathcal{P}'}$ and $\mathbb{L}_{\mathcal{P}''} = \mathbb{L}_\mathcal{P} + \mathbb{L}_{\mathcal{P}'}$.

We define a distance function $DL(\mathbb{L}, \mathbb{L}')$ to compute the distance between two policy specifiers $\mathbb{L}$ and $\mathbb{L}'$. Let $l \in \mathbb{L}$, $l' \in \mathbb{L}'$, and let I be the set of all node selection criteria *ID*s in the union of $\mathbb{L}$ and $\mathbb{L}'$. For each $i \in I$ let $d_i = (k_l - k_{l'})^2$; $k_l = 0$ or $k_{l'} = 0$ if their respective *ID* does not exist. Finally, $DL() = \sqrt{\sum d_i}$

The function $\mathsf{MIN\_POLICY}(x, y)$ is used to compare the characteristics of two policies and return the policy with the smallest characteristics. The values x and y are values such as *load* or $|N|$.

```
1   FOR EACH s ∈ S
2       FOR EACH l ∈ 𝕃
3           IF ( k_l > 0 AND C_s ⊆ 𝕂_l )
4               k_l = k_l − 1
5               ADD s TO nodeset
6       IF ( k_l == 0, ∀ l ∈ 𝕃 )
7           RETURN nodeset
8   RETURN NULL
```

Figure 2: Create a new policy by using only singletons.

```
FOR EACH 𝒫 ∈ E_𝒫
    IF ( 𝕃_𝒫 ⊆ 𝕃 )
        O = MIN( |N|_Q, |N|_𝒫 )
        Q = MIN( load_O, load_𝒫 )
load_Q = load_Q + 1
RETURN Q
```

Figure 3: Create a new policy by reusing an existing policy.

3.2 Policy creation

The algorithm for creating policies follows the general recipe presented in Section 2 and consists of three phases. The input is a policy specifier $\mathbb{L}$, a set of existing policies $E_\mathcal{P}$, and the topology $\mathcal{T}$, the output is a set of nodes that satisfies $\mathbb{L}$.

In the first phase (Figure 2) the algorithm attempts to satisfy the policy using only singleton nodes. The input is a set S of singleton nodes and the policy specifier $\mathbb{L}$. The output is a set of nodes if the policy is satisfiable. The variable *nodeset* stores the nodes selected for the policy. The algorithm iterates through the set of singletons. A singleton node is added to the *nodeset* (line 5) if it matches one of the node characteristics listed in the policy, and the required number of nodes has not yet been reached. The complexity of this phase is $O(|S| \cdot |\mathbb{L}|)$; in general, $|\mathbb{L}|$ tends to be small, less than ten entries, thus the algorithm is mostly linear in the number of singletons.

If $\mathbb{L}$ cannot be satisfied with singletons, the second phase of the algorithm (Figure 3) is executed which attempts to satisfy $\mathbb{L}$ with an existing policy $\mathcal{P}$. The input

1 FOR EACH s $\in S$
2   FOR EACH $l \in \mathbb{L}$
3   IF ($\mathcal{C}_\mathsf{s} \subseteq \mathbb{K}_l$)
4   ADD s TO $\mathcal{R}$
5 FOR EACH $\mathcal{P} \in E_\mathcal{P}$
6     $\mathcal{Q} = \mathcal{P} + \mathcal{R}$
7   IF ($\mathbb{L}_\mathcal{Q} \subseteq \mathbb{L}$)
8         $\mathcal{O}$ = MIN_POLICY($|N_\mathcal{O}|$, $|N_\mathcal{Q}|$)
9 IF ($\mathcal{O}$)
10   EXTRACT SELECTED s $\in \mathcal{R}$, ADD TO *nodeset*
11   RETURN $\langle \mathcal{P}, nodeset \rangle$
12 FOR EACH $\mathcal{P} \in E_\mathcal{P}$
13     $\mathcal{Q}$ = MIN_POLICY($DL(\mathbb{L}, \mathbb{L}_\mathcal{Q})$, $DL(\mathbb{L}, \mathbb{L}_\mathcal{P})$)
14 ADD $\mathcal{Q}$ TO *policyset*
15 LOOP
16   FOR EACH $\mathcal{P} \in E_\mathcal{P} \backslash policyset$
17         $\mathcal{O} = policyset + \mathcal{P}$
18         $\mathcal{Q} = \mathcal{O} + \mathcal{R}$
19   IF ($\mathbb{L}_\mathcal{O} \subseteq \mathbb{L}$)
20             $\mathcal{U}$ = MIN_POLICY($|N_\mathcal{U}|$,$|N_\mathcal{O}|$)
21   IF ($\mathbb{L}_\mathcal{Q} \subseteq \mathbb{L}$)
22             $\mathcal{V}$ = MIN_POLICY($|N_\mathcal{V}|$,$|N_\mathcal{Q}|$)
23         $\mathcal{Q}$ = MIN_POLICY($DL(\mathbb{L}, \mathbb{L}_\mathcal{Q})$, $DL(\mathbb{L}, \mathbb{L}_\mathcal{O})$)
24   IF ($\mathcal{U}$)
25   RETURN $\langle \mathcal{U}$, NULL$\rangle$
26   IF ($\mathcal{V}$)
27   EXTRACT SELECTED s $\in \mathcal{R}$, ADD TO *nodeset*
28   RETURN $\langle \mathcal{V}, nodeset \rangle$
29   ADD $\mathcal{Q}$ TO *policyset*
30   IF ($E_\mathcal{P} \backslash policyset = \emptyset$)
31   RETURN $\langle$NULL, NULL$\rangle$

Figure 4: Create a new policy by combining existing policies and singletons.

is a set $E_\mathcal{P}$ of existing policies and the policy specifier $\mathbb{L}$. The output is a reference to an existing policy $\mathcal{P}$. Each existing policy $\mathcal{P}$ is compared to the policy specifier $\mathbb{L}$ (line 2). The idea is to select $\mathcal{P}$ such that $\mathbb{L}_\mathcal{P} \subseteq \mathbb{L}$ and $\mathcal{P}$ has the smallest number of nodes (line 3). Selecting policies of similar size improves the load balancing because the load is better distributed. Otherwise, a large policy would satisfy both large and small policies causing it to be overloaded while under-utilizing the smaller policies. If two selections are equal then the policy with the smallest load is selected (line 4). The complexity of this phase is $O(|E_\mathcal{P}|)$.

If no existing policy satisfies $\mathbb{L}$ then the third phase (Figure 4) is executed. To satisfy $\mathbb{L}$ the algorithm first attempts to find the closest matching existing policy and extend it using singletons. Second, it attempts to combine several existing policies, and third it attempts a combination of the two. The input is a set S of singletons, a set $E_\mathcal{P}$ of existing policies, and the policy specifier $\mathbb{L}$. The output is a set of existing policies and singletons that satisfy $\mathbb{L}$. First, the algorithm creates a policy $\mathcal{R}$ by combining all of the available singleton nodes' node characteristics (lines 1-4). In this way the contribution or the extension provided by the singletons is easily computed. Second, we determine if there exists an existing policy $\mathcal{P}$ such that when extended by the singletons it is going to satisfy $\mathbb{L}$ (lines 5-8). If the first part is successful (lines 9-11) then we extract the singletons that form the extension to $\mathcal{P}$ (line 10) and return the existing policy $\mathcal{P}$ and the set of singletons in *nodeset*. If the first part is unsuccessful we determine $\mathcal{P}$ that partially satisfies $\mathbb{L}$ the best (lines 12-13), i.e. $DL(\mathbb{L}_\mathcal{P}, \mathbb{L})$ is minimized, as the starting point for the second part.

In the second part (lines 15-31) we loop through the existing set of policies looking for three types of policy combinations. Given the current set of policies, *policyset*, one, is there an existing policy $\mathcal{P}$ such that $policyset + \mathcal{P}$ satisfies $\mathbb{L}$ (lines 19-20), two, is there a set S of singletons such that $policyset + \mathcal{P} + S$ satisfies $\mathbb{L}$ (lines 21-22), and three is there $\mathcal{P}' = policyset + \mathcal{P}$ such that $\mathbb{L}$ is not satisfied but $DL(\mathbb{L}_\mathcal{P}, \mathbb{L})$ is minimized (line 23). If there exists a simple combination of policies that satisfies $\mathbb{L}$ (lines 24-25) then that set of policies is returned. Else if there exists a simple combination of policies plus singletons that satisfies $\mathbb{L}$ (lines 26-28) then that set of policies and singletons are returned. Otherwise the loop iterates until there are no more existing policies to combine. The algorithm takes a a greedy approach to constructing the combination of policies and singletons to satisfy $\mathbb{L}$; the optimal approach is exponential. The runtime of this phase is $O(|E_\mathcal{P}|^2)$.

3.3 Replica node selection

The algorithm for selecting a replacement for a failed node also consists of three phases and follows the general recipe presented in Section 2. The input is the failed node n, the node selection criteria $\mathbb{K}$, a set of existing policies $E_\mathcal{P}$, and the topology $\mathcal{T}$, the output is one or more nodes that replace the failed node n.

In the first phase (Figure 5) the algorithm attempts to replace the failed node n with one or more singleton nodes; more than one singleton is used if n is a cut vertex. The benefit of replacing n with multiple singleton nodes is that the topology is improved by splitting a large component

```
components = FindCutVertex( n )
IF ( |components| == 1 )
    components = FindAuxilaryCutVertex( n )
FOR EACH s ∈ S
    IF ( C_s ⊆ 𝕂 )
        ADD s TO match
IF ( |match| == 0 )
    RETURN failure
SORT( components )
FOR EACH c ∈ components
    IF ( |match| > 0 )
        match = match\m,  m ∈ match
    ReplaceNode( c, n, m )
RETURN success
```

Figure 5: Find replacement nodes using only singletons.

into several smaller ones and thus, reducing the number of inter-connections. In addition, the algorithm tests for a vertex n′ such that if n and n′ fail then the component would become disconnected; we call n′ an *auxiliary cut vertex*. This is done in anticipation of node n′ failing, so that when n′ does fail the component can be split into several smaller components.

Let T be the connected component that n was part of. Let S be the set of singletons. First the algorithm determines if n is a cut vertex, if it is not, the algorithm proceeds to find an auxiliary cut vertex n′; the result is one or more components. Next, the algorithm iterates through the available singletons and selects a number of nodes, up to the number of components, that satisfy the node selection constraint $\mathbb{K}$. If no singletons are available the algorithm proceeds to the second phase.

The components are then sorted from largest to smallest in order to split off the largest components first; thus maximally reducing the overall size of the components. This is especially important when the number of available singletons is less than the number of components. For each component, n is replaced by one of the selected singleton nodes. When there is only one selected singleton left, it is used to replace n in the remaining components. Determining the cut vertex and the auxiliary cut vertex dominates the runtime of this phase. To determine the cut vertex requires $O(|T|)$. To determine the auxiliary cut vertex it is necessary to iterate through all the nodes in T, removing each one and running the cut vertex algorithm, thus the complexity is $O(|T|^2)$.

```
1   FOR EACH 𝒫 ∈ E_𝒫
2       U = ⋃ R_𝒫
3   A = T − U
4   FOR EACH a ∈ A
5       IF ( C_a ⊆ 𝕂 )
6           node = MIN_NODE( load_node, load_a )
7   RETURN node
```

Figure 6: Find a replacement node within the component.

```
FOR EACH g ∈ G
    FOR EACH r ∈ g
        IF ( C_r ⊆ 𝕂 )
            node = MIN_COMPONENT( |g_node|, |g_r| )
RETURN node
```

Figure 7: Find a replacement node by combining components.

When no singletons are available to replace the failed node n an attempt is made to find a replacement node from amongst the nodes in the component T. This is done in the second phase (Figure 6) to increase utilization and to load balance. Although components would grow smaller if done in the first phase, each component would be responsible for storing more data and the singletons would be under utilized. Let T be the connected component that n was part of. Let $E_{\mathcal{P}}$ be the set of policies that node n was responsible for. Let R be the set of nodes listed in a policy $\mathcal{P} \in E_{\mathcal{P}}$. The input is the failed node n, the node selection criteria $\mathbb{K}$, and T, the output is the replacement node.

First the nodes that cannot be selected as a replacement for n are determined. These nodes are easy to categorize as those that are adjacent to n and are determined by taking the union of all the nodes listed in all of the policies that the node n was responsible for (lines 1-2) and subtracting it from T (line 3). Next the algorithm iterates through A and selects nodes that satisfy the node selection criteria $\mathbb{K}$. If there are multiple possible selections then the node with the smallest *load* is selected. The complexity of this phase is $O(|T|)$.

```
FOR EACH g ∈ G
    FOR EACH r ∈ g
        components = FindCutVertex(r)
        IF ( |components| > 1 )
            match = NULL
            FOR EACH s ∈ S
                IF ( C_s ⊆ K_r )
                    ADD s TO match
            IF ( |match| > 0 )
                SORT( components )
                FOR EACH c ∈ components
                    IF ( |match| > 0 )
                        match = match\m, m ∈ match
                    ReplaceNode( c, r, m )
                RETURN success
```

Figure 8: Periodic maintenance.

The third phase (Figure 7) is executed only when the first two phases have failed to select a replacement node for n. In this final phase the algorithm looks at joining two components. The input to the algorithm is the failed node n and a set of components G, the output is a replacement node for n. This phase is performed as a last resort because it worsens the topology by joining components, making them bigger and reducing their number; contrary to our initial goals. In this final phase we iterate through all of the available nodes r that are listed in the components $g \in G$ to locate a node that can replace n. The node that replaces n should also be part of the smallest component so that the increase in the size of the resulting component is as small as possible. The complexity of this phase is $O(|\mathcal{T}|)$ since it iterates through all of the nodes in the system.

3.4 Periodic maintenance

Apart from selecting nodes for policies and finding replacement nodes, we present an additional algorithm (Figure 8) to perform periodic maintenance to improve the topology. This algorithm is similar in nature to the first phase of the algorithm presented in Section 3.3 which selects replacement nodes. The algorithm finds cut vertices and splits components into smaller components. The input to the algorithm is a set S of singletons and a set G of components. The algorithm iterates through the nodes r in all of the components $g \in G$ to find a cut vertex v. Once one is found, the algorithm iterates through the singletons to see if there is at least one singleton s that is comparable to the cut vertex v. One singleton is sufficient since at least one component changes from using v to s and the others can continue to use v. Once the singletons are selected the process of component splitting is performed, similar as in the first phase of the replica node selection algorithm (Section 3.3. The complexity for this part is $O(|\mathcal{T}| \cdot |\mathsf{r}|)$.

4 Results

Simulation is used to evaluate the policy creation and replica node selection algorithms presented above. The objective is to evaluate the effectiveness of the algorithm in maintaining the topology as the system evolves over time; i.e., nodes fail, new nodes arrive, and new replication policies are created. In the evaluation we show that the algorithm performs better than simple naive approaches, that it is scalable, that the algorithm is insensitive to the rate of policy creation, and that it is insensitive to the size of the system.

4.1 Simulation

A simple discrete event simulator is used to perform the simulations. There are four events that are generated in the simulation: node failure, new node insertion, policy creation, and periodic maintenance. There are several different classes of nodes. The simulation assumes that nodes of the same class are independent, have similar failure modes, and that their lifetimes are exponentially distributed; different classes have different failure modes and different expected lifetimes. For certain peer-to-peer systems this assumption has been shown to be valid [3]. Although not perfect, we believe that the assumption of exponentially distributed lifetimes represents a valid approximation, especially given that we are interested in the evolution of the system rather than specific events.

The probability of a node failure during a period of length τ is $1 - e^{-\mu\tau}$, where $1/\mu$ is the expected lifetime of a node. In our simulation there are two classes of nodes having an expected lifetime of 100 and 300 days; higher expected liftetimes translate to more reliable nodes. Periods of 100 and 300 days were chosen because a high frequency of events enables us to better evaluate the evolution of the system, which we are predominantly interested in (although the lifetimes may seem short, they are somewhat based on the upgrade cycle in our department). We used a similar distribution for policy creation with $1/\mu$ being varied between one and 120 days. The number of policies created at each policy creation event varied uniformly between one and five.

Each policy specifies that data be stored on three to seven low reliability (100 days) nodes and one to four high reliability (300 days) nodes. Although Starfish [8] showed that three replicas is sufficient, it also assumes that at least two of the three replicas are tightly coupled to the primary. We do not make such assumptions and thus increase the number of replicas to achieve the same or greater level of protection.

After each event the state of the topology is recorded. Specifically, the total number of nodes, the number of components, and the number of singletons in the system. In addition, the minimum, the average, and the maximum node degrees are computed; the same statistics are computed for the number of distinct policies on a node. This information is used in the metrics, Section 4.2, to compute the quality of the topology.

4.2 Metrics

To evaluate the system topology we employ three metrics, the average node degree, the average number of distinct policies per node, and the number of components in the system. The node degree is equivalent to the vertex degree. The degree d of a node a is the number of nodes it is linked to, and represents the overhead, in terms of the number of nodes affected if a fails and the number of messages, $O(d)$, necessary to perform the recovery. The smaller the degree, the fewer nodes affected by the failure of a. This metric measures the effectiveness of both the policy creation and the replica node selection algorithms.

The number of distinct policies on a node is representative of its virtual load. A node with a large number of distinct policies is responsible for a larger amount of data and must interact with a potentially larger number of nodes to keep the data safe. Thus, the goal is to maintain the average number of distinct policies per node small, and ensure that the policies are distributed across all the nodes. A node that is responsible for many reused policies must still manage more data, but does not incur more overhead with respect to policy management, since two identical policies are managed as one policy that simply encompasses more data. This metric predominantly measures the effectiveness of the policy creation algorithm since the majority of the load on a node is instantiated at policy creation time.

The last metric is the number of components in the system. This metric is an overall measure of how well the policy creation and the replica node selection algorithms are working together. When the number of components is large then the individual components are relatively small, the node degrees are small, and a failure does not affect a large number of nodes. In addition, the majority of nodes are participating in replicating data. A smaller number of components means that either only a few nodes are participating in policies, and thus some nodes are heavily loaded while others are idle, or the components are large, the number of inter-connections between nodes is large, and thus a node failure may affect a large number of nodes.

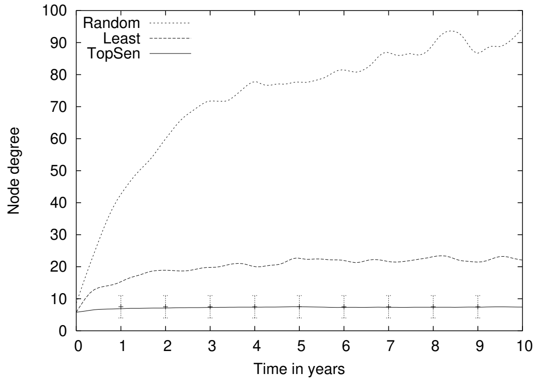


Figure 9: Average node degree.

In Section 3 we presented two algorithms, one for policy creation and one for replica node selection. Unless explicitly mentioned otherwise, they are evaluated as a single entity.

4.3 The three algorithms

As part of the evaluation of our TopSen algorithm it was compared to two other algorithms, *Random* and *Least*. The *Random* algorithm selects a random node of the same type when replacing a failed node and selects the nodes at random when creating a policy. The *Least* algorithm selects a node of the same type that has the smallest node degree to replace a failed node, and selects the nodes with the smallest degrees when creating a new policy. If there is a tie, the node with the smallest number of policies is selected. The simulation consists of 1k nodes, over a ten year period, and initially stores 100 policies (10% of the nodes). The initial number of policies may seem low but this is necessary to demonstrate the decline of the number of components in Figure 11.

Figure 9 presents the average node degree for the three algorithms. The error-bar markers denote the optimal range, defined by $[|\mathcal{P}|_{min}, |\mathcal{P}|_{max}]$, for the average node degree which occurs when the number of nodes in a component equals to the number of nodes listed in a policy.

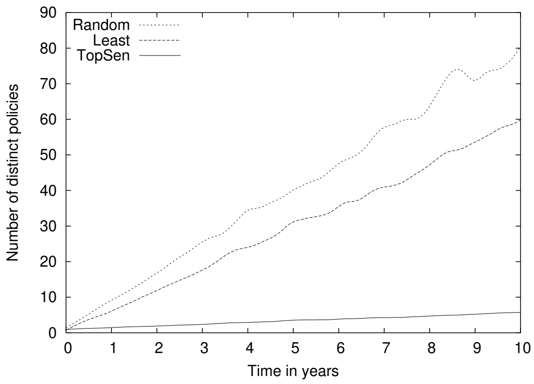


Figure 10: Average number of distinct policies per node.

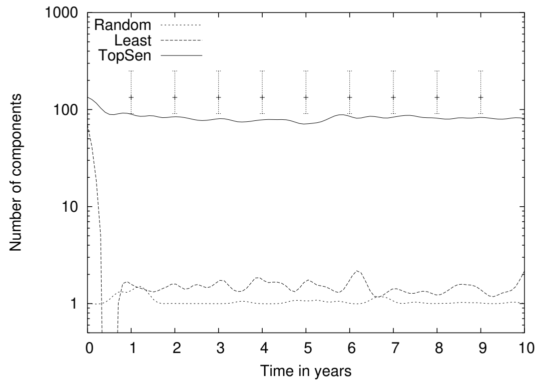


Figure 11: Number of components.

Of the three algorithms, the TopSen algorithm maintains the lowest, the most consistent, and uniform node degree throughout the simulation. In both the Random and the Least algorithms the node degree increases over time. After a few years the average node degree is about ten times higher for the Random algorithm and three to four times higher for the Least algorithm. That is, compared to the TopSen algorithm, ten times as many nodes are affected by a node failure with Random selection, and three to four times as many nodes are affected by a node failure with Least selection. Both Random and Least algorithms create topologies with higher average node degrees because they do not attempt to reduce the number of interconnections made when replacing failed nodes. The reason the Random algorithm has the highest node degree is that when a node is added to a policy, its node degree increases by the size of the policy. Thus, selecting a node at random, and adding it to a policy, increases the node's degree (on average) by the average size of a policy. And this increases the average node degree much more than selecting a low degree node (Least) or a node with the majority of the inter-connections already in place (TopSen) and inserting it into an average sized policy.

Figure 10 shows the average number of distinct policies per node for the three algorithms. The TopSen algorithm performs best since it tries to overlap and reuse existing policies as much as possible. Thus, the number of completely new policies, with new inter-connections, created is significantly less than that of the Random and the Least algorithms. The Random and the Least algorithms create unique new policies significantly more often because they do not take into consideration the underlying topology. In addition, the TopSen algorithm has the ability to split components thus further reducing the node degree and the number of distinct policies on a node.

Figure 11 presents the number of components, on a log-scale, in the system for the three algorithms. The error-bar markers denote the optimal range, defined by $\left[\frac{|\mathcal{T}|}{|\mathcal{P}|_{min}}, \frac{|\mathcal{T}|}{|\mathcal{P}|_{max}}\right]$, for the number of components which occurs when the number of nodes in a component equals to the number of nodes listed in a policy. The TopSen algorithm does significantly better than the Random or the Least algorithm. Whereas with the Random and the Least algorithms the number of components quickly drops down to one or two, the TopSen algorithm maintains the number of components at a relatively constant level; the discontinuity for the Least algorithm is due to the steepness of the curve and the data smoothing function, there is always at least one component. This result is in-line with the other two metrics and confirms that the TopSen algorithm is maintaining the topology better than the Random or the Least approach, with the result of having many small disconnected components.

4.4 Scalability

The scalability of TopSen is compared to the Random and the Least algorithm by performing simulation runs of 100, 1k, and 10k nodes, for a 10 year period, with an initial policy load of 10% of the number of nodes.

Figure 12 demonstrates that the TopSen algorithm scales considerably better than the Random algorithm and noticeably better than the Least algorithm with respect to the node degree. For all 100, 1k, and 10k node simulation runs the node degree is low, about six, and constant. The TopSen algorithm is not affected by the number of nodes in the system since its focus is strictly at the topology and component level. No matter how many nodes are available it always selects a node that induces the fewest number of new inter-connections in the system. The Least

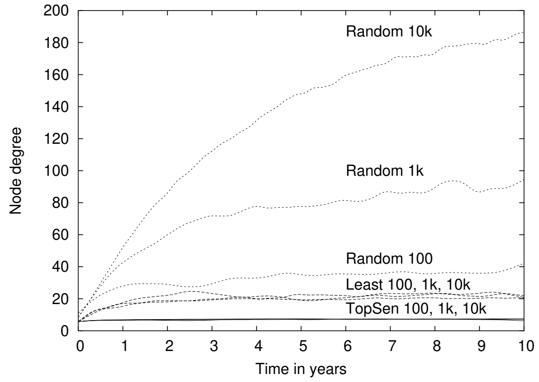


Figure 12: Average node degree in 100, 1k, and 10k node simulations.

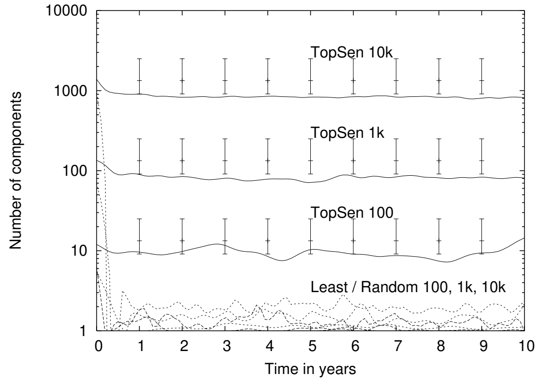


Figure 13: Number of components in 100, 1k, and 10k node simulations.

algorithm is the next best performer. It maintains the node degree relatively constant but it is about three to five times higher than that of the TopSen algorithm. Since the Least algorithm uniformly spreads the policy load by always selecting a node with the smallest node degree, the performance of the algorithm slightly improves as system size increases because there is a larger selection of replica nodes. The Random algorithm performs the worst since it pays no attention to the node degree or the topology.

For the Random and the Least algorithms the node degree and the number of distinct policies are directly affected by the number of nodes in the system and the rate of policy creation. If the number of nodes in the system is small and the policy creation rate is high then both the node degree and the number of distinct policies will also be large. This is due to the Random and the Least algorithms being insensitive to the underlying topology. In Section 4.5 we discuss the affects of policy creation rate on the system topology. The TopSen algorithm is sensitive to the system topology and attempts to overlay and reuse existing policy definitions as much as possible. Thus, TopSen, is practically insensitive to the number of nodes in the system and the policy creation rate.

The scalability advantages of TopSen are seen most clearly when looking at the number of components in the system. TopSen maintains the number of components relatively constant and higher than the Random and the Least algorithms (Figure 13). For the TopSen algorithm the average number of components is about an order of magnitude less than the number of nodes in the system. In the Random and the Least algorithms the number of components drops to just a couple, irrespective of the number of nodes in the system. Again, the TopSen algorithm's superiority comes from being sensitive to the topology and avoiding the creation of new inter-connections between nodes.

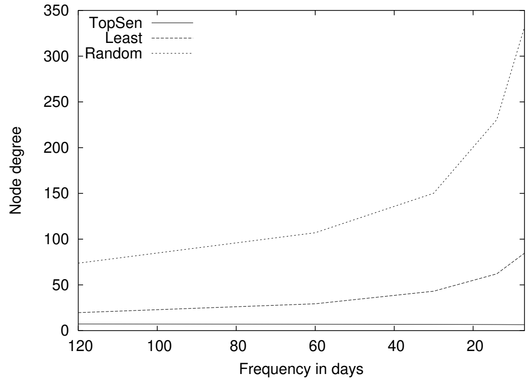


Figure 14: Average node degree vs. policy creation frequency.

4.5 Policy creation

Next, we examine how the frequency of policy creation affects the policy creation and replica node selection algorithms. The frequency of policy creation is varied from 120 days to seven days between policy creation events. For each frequency a simulation run is performed with 1k nodes for a period of ten years.

We found that the policy creation frequency does not have a significant affect on the topology when the TopSen algorithm is used for replica selection compared to the Random or the Least algorithms. The average node degree varied slightly (Figure 14). Although, the rate at which the number of distinct policies increases on a node approximately doubled, from 0.74 to 1.21 per year, the increase

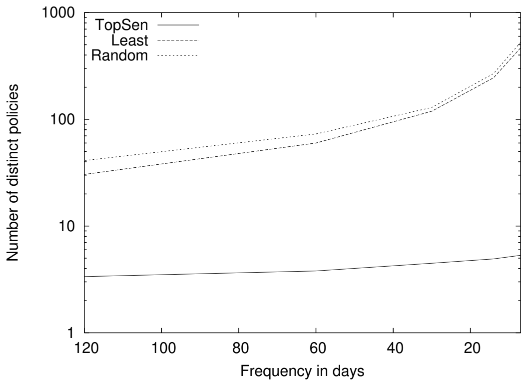


Figure 15: Average number of distinct policies per node vs. policy creation frequency.

is still small given that the frequency of policy creation increased by two orders of magnitude; compared to the other two algorithms (Figure 15) we do significantly better. We found no strong correlation between the number of components and the frequency of policy creation.

4.6 Splitting and Periodic maintenance

We now examine the benefits of splitting components. Splitting is attempted when a node has failed and is a cut vertex and during periodic maintenance when a cut vertex is encountered. No perceivable benefits were observed in terms of the node degree and the number of distinct policies per node; we only look at the number of components in the system.

We use the result of no splitting of any type as the baseline for our measurements. With no splitting there are on average 80 components and the standard deviation is approximately 6.9. Just performing the splitting during a failure improves the average number of components by 7.5%. Benefits from the periodic maintenance are seen only if it is performed every 14 days or less; 12.5% improvement if done weekly, 23.8% improvement if done every two days, and 35.0% improvement if done daily. In addition, the frequency at which benefits are seen from the periodic maintenance is dependent on the expected lifetimes of the nodes. The longer the lifetime the smaller the frequency necessary to see benefits from periodic maintenance.

5 Application

For the simulation we assumed that nodes have complete knowledge of the system. This is a reasonable assumption for two reasons. First, systems such as Farsite [1] assume that there is complete knowledge. Other systems such as Pangaea [21] and PDR [4] actively discover nodes in the system, usually using a gossip style protocol [23], thus eventually arriving at a relatively complete knowledge of the system. Second, the algorithm requires no modifications to operate in systems such as Pangaea and PDR because discovering a node, or a set of nodes, is equivalent to node insertion events. Thus, for these systems the whole system can be viewed as a collection of smaller disconnected topologies that at some point merge to form bigger topologies. The algorithms operate identically whether the nodes have complete knowledge or discover it eventually. The system is assumed to have *fail stop* semantics and transient node or network failures are not the responsibility of the TopSen algorithm.

6 Related work

A large body of work exists in the area distributed hash table (DHT) based peer-to-peer storage systems [7, 11, 14, 20, 22]. The main benefit of this approach is the ability to locate file data using $O(\log N)$ messages among N peer nodes. Each node is assigned a quasi-random ID; similarly, files (or blocks) are assigned an ID taken from the same space and are stored on nodes with numerically closest IDs. The key to tolerating failure is the random assignment of node IDs; this ensures that nodes with similar IDs are unrelated to each other in the underlying network topology. Data replicated on nodes with consecutive IDs is thus reasonably invulnerable to a single point of failure. In principle a file's (or block's) location is randomly chosen and fixed when it is first added to the system. Thus, DHT based systems assume that the cost of replicating to a node and its reliability is uniform across the system, and that its failure modes are independent from others. Although, this may be true in an ideal world, these assumptions are generally over simplified [13, 24].

Other peer-to-peer systems [1, 4, 8, 21] take a different approach to selecting nodes. Farsite [1] designates a subset of nodes to store a subtree of the file system. These subsets are non-overlapping and are initially chosen at random. In addition, Farsite tracks and continuously moves metadata and file data from low-availability to higher availability machines increasing its availability and reliability. Farsite is designed to operate in a LAN environment, it assumes that nodes are connected by a high bandwidth, low latency network, and are homogeneous and independent. Although, these assumptions are mostly acceptable in a closed local environment, they make Farsite infeasible in the wide area.

Pangaea [21] stores files on nodes that frequently access them. A subset of these nodes, with higher availability and lower latencies, are charged with ensuring that the file's replication level is maintained. Each node has a membership module that keeps track of a list of regions, the round trip time between each region, and the status of other nodes including their available disk space, availability, and their location. Pangaea uses this information to select nodes to store replicas and to find replacements for failed nodes. The topology is exchanged using a gossip style protocol [23].

Starfish [8] predominantly uses monitoring to ensure that replicas are alive. This system requires intervention by the administrator to specify replacement replicas and to induce a switch over from a failed to a working replica. They also demonstrated that three replicas are sufficient to achieve a reasonable level of protection. Two of the replicas have to be tightly coupled to the primary by a high bandwidth connection and one of the two has to be offsite. The third replica has to be offsite but does not need to be tightly coupled to the primary.

PDR [4] allows users to define policies that specify what files, and where and when these files are replicated. The system treats these policies as invariants, and maintains the replication level, or policies, through monitoring and re-replication. Topology is disseminated between the nodes using a gossip style protocol [23]. PDR employs the TopSen algorithm to select nodes when creating replication policies and when selecting replacements for failed nodes.

7 Conclusion

In this paper we presented TopSen, an algorithm for selecting replica nodes in a peer-to-peer replication system. The algorithm is sensitive to the system topology and to other node characteristics such that the results of the algorithm minimizes the cost of recovery from a failed replica with respect to the number of messages and bandwidth. This is accomplished by maintaining a topology where the number of inter-connections between nodes is minimized and new connections are only added as a last resort. Thus, a failure of a node only affects a small number of other nodes.

In addition, topology sensitive replica node selection aids system scalability. If replica selection is not done wisely then the number of nodes involved in the recovery from a single node failure can be very large. In a system of a million nodes, where the average uptime is 100 days, there are on average about 10k failures per day. Given that many distributed consensus and election algorithms require $O(n^2)$ messages, where n is the number of nodes, it could require up to 10^{16} messages simply to determine who should lead the recovery process and what needs to be re-replicated; to handle a failed node requires $O(n^2)$ messages. For a system of 10k nodes approximately 10^{10} messages would be needed, plus the data to replicate. Thus, these systems would spend all their time performing recovery.

We have shown that through topology sensitive replica node selection it is possible to maintain the topology such that the number of messages necessary to perform recovery is constant. That is, the cost of recovery is independent of the size of the system.

Many distributed systems assume that nodes are identical, independent, and have similar failure characteristics. This may be a reasonable assumption to make in an idyllic world, but it has been shown that such assumptions are not applicable in reality [13, 24]. This is particularly true with replication systems where node characteristics, such as security, reliability, bandwidth, and failure modes must be considered to ensure that data is secure and available.

Replication systems are becoming more distributed. In addition, given the ever increasing amounts of data we are creating, these replication systems are less likely to uniformly replicate all of it. Although the physical storage to store this data is cheap, the pipes connecting this storage are not, and are unable to handle the volume [17]. Given that all data is not created equal, there is no need to replicate it uniformly. Thus, replication systems need to selectively replicate data to minimize cost, and they must select the location of the replicas such that the required level of protection is obtained.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systans Design and Implementation (OSDI-02)*, Dec. 2002.

[2] R. Anderson. The eternity service, 1996.

[3] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of IPTPS'03*, 2003.

[4] D. Brodsky, A. Brodsky, M. Feeley, and N. Hutchinson. Policy driven replication. Technical Report TR-2003-15, University of British Columbia, 2003.

[5] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[6] B. Cohen. Incentives build robustness in bittorrent, May 2003.

[7] F. Dabek, M. F. Kasshoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, Oct. 2001.

[8] E. Gabber, J. Fellin, M. Flaster, F. Gu, B. Hillyer, W. T. Ng, B. Ozden, and E. Shriver. Starfish: highly-available block storage. In *Proceedings of the FREENIX Track: 2003 USENIX Anual Technical Conference*, pages 151–163, June 2003.

[9] Gnutella. http://www.gnutella.com.

[10] Kazaa. http://www.kazaa.com.

[11] J. Kubiatowicz, D. Bindel, Y. Chen, 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 *Proceedings of ACM ASPLOS*, Nov. 2000.

[12] M. Mahalingam, C. Tang, and Z. Xu. Towards a semantic, deep archival file system. In *The 9th International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003)*, 2003.

[13] K. Marzullo. Theory and practice for fault-tolerant protocols on the internet, Oct. 2002.

[14] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systans Design and Implementation (OSDI-02)*, Dec. 2002.

[15] Napster. http://www.napster.com.

[16] Overnet. http://www.overnet.com.

[17] D. A. Patterson. A conversation with Jim Gray. *ACM Queue*, 1(4):53–56, June 2003.

[18] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Conference on File and Storage Technologies (FAST '02)*, Mar. 2002.

[19] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the oceanstore prototype. In *Conference on File and Storage Technologies (FAST '03)*, Mar. 2003.

[20] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer stroage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, Oct. 2001.

[21] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systans Design and Implementation (OSDI-02)*, Dec. 2002.

[22] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 172–183, Dec. 1995.

[23] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of Middleware*, 1998.

[24] H. Weatherspoon, T. Moscovitz, and J. Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proceedings of International Workshop on Reliable Peer-to-Peer Distributed Systems*, pages 362–367, Oct. 2002.