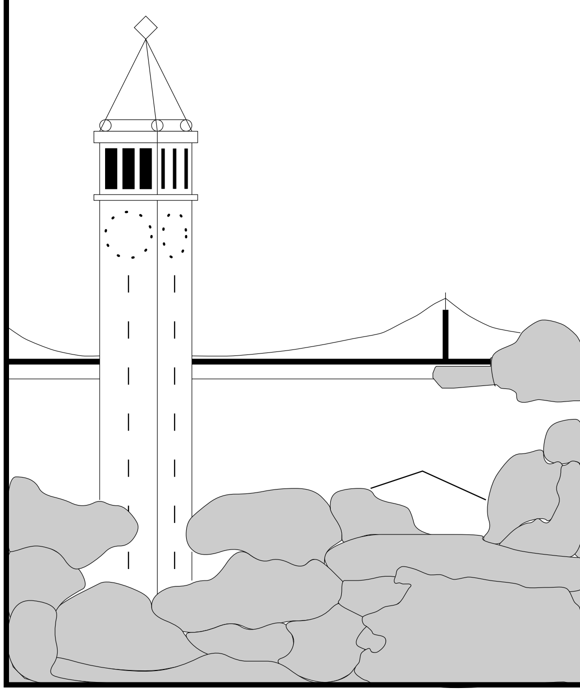
Long-Term Data Maintenance in Wide-Area Storage Systems: A Quantitative Approach

Hakim Weatherspoon, Byung-Gon Chun, Chiu Wah So, John Kubiatowicz
University of California, Berkeley



Report No. UCB/CSD-05-1404

July 2005

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Long-Term Data Maintenance in Wide-Area Storage Systems: A Quantitative Approach

Hakim Weatherspoon, Byung-Gon Chun, Chiu Wah So, John Kubiatowicz
University of California, Berkeley

July 2005

Abstract

Maintaining data replication levels is a fundamental process of wide-area storage systems; replicas must be created as storage nodes permanently fail to avoid data loss. Many failures in the wide-area are transient, however, where the node returns with data intact. Given a goal of minimizing replicas created to maintain a desired replication level, creating replicas in response to transient failures is wasted effort. In this paper, we present a principled way of minimizing costs while maintaining a desired data availability. Design choices include choosing data redundancy type, number of replicas, extra redundancy, and data placement. We demonstrate via trace-driven simulation that significant maintenance efficiency gains can be realized in existing storage systems with the correct choice of strategies and parameters. For example, we show that DHash can reduce its costs by a factor of 31 while maintaining the same desired data availability.

1 Introduction

Maintaining data replication levels is a fundamental process of wide-area storage systems. Wide-area storage systems replicate data to reduce risk of data loss and continually trigger data recovery by replacing lost replicas as nodes fail. We call this process data maintenance. It includes data redundancy schemes, node selection for data placement, monitoring node availability using heartbeats, detecting node failures by missing heartbeats and triggering data recovery.

Challenges arise when configuring a storage system for efficient data maintenance. The bandwidth costs due to data maintenance choices are affected by node availability and failure characteristics. For instance, transient node failures where nodes return from failure with data intact, can increase the cost of maintaining data availability significantly. Additionally, correlated failures can compromise data availability. Figure 1 shows an example of the importance of data maintenance. In this example, a replication scheme produced eight total replicas for some object and seven replicas were required to be available to satisfy some data availability constraint. Using a data placement strategy, replicas were then distributed throughout the wide-area. Over time, a replica in Georgia permanently failed, data was lost, and a replica in Washington transiently failed since only a heartbeat was lost (Figure 1(a)). As a result of the failures, two new nodes in Arizona and Minnesota were chosen and replicas were copied to the new nodes (Figure 1(b)), we call this data recovery. Notice that the node in Washington could return from failure with data intact, in which case, recovery work would have been wasted creating a new replica.

Many systems effectively use simulation to explore the cost tradeoffs of data maintenance; however, complete explorations via simulation have been limited in literature and there are few studies to compare different storage systems. In general, simulation can provide a way to study system design alternatives in a controlled environment, explore system configurations that are difficult to physically construct, observe interactions that are difficult to capture in a live system, and compare the cost tradeoffs over time. For example, Total Recall [5] showed via simulation that lazy repair can mask transient failures, delay triggering data recovery and reduce the cost of data maintenance. However, the simulations have been on the order of weeks which is insufficient to measure the long-term costs. Moreover, many studies use high-churn P2P file-sharing traces [5, 6, 22], which represent environments different from storage system environments. Additionally, the growth of systems (i.e., the increase of maintained data in the system) has not been considered. Finally, most systems assume failures are independent without any quantification. We address these issues by presenting a unified view of storage systems and evaluating systems with a long-term PlanetLab trace.

Wide-area storage systems can be realized with a specific instantiation of data redundancy type, data recovery style, and data placement strategy. Many systems choose a specific set of strategies and parameter values, but it is not clear why systems choose such values. Surprisingly, we found out that the specific setting of Pond [25] guarantees only two 9's of data availability in PlanetLab [2], and that of Dhash [12] is configured to spend very high maintenance bandwidth. These are all consequences of unprincipled approaches of parameterizing storage systems for long-term data maintenance.

In this paper, we show that significant efficiency gains can be realized in existing storage systems with analysis and trace-driven simulation of maintaining data on a long-term basis. In particular, we explain a principled way of choosing data redundancy type, number of replicas, recovery style, and data placement. Our analysis is based on PlanetLab trace, one of the largest open wide-area systems, and compare many existing storage system parameterization and their long-term costs on PlanetLab.

The contributions of this work are as follows:

- We present a unified view of existing wide-area storage systems in terms of data redundancy type, data recovery style, and data placement. We evaluate the long-term maintenance

(a) Threshold Reached (b) Data Recovered

Figure 1: Example. Maintaining Data in a Wide Area Storage System.

costs of the systems using a trace-driven simulation of PlanetLab. In particular, we evaluate the specific parameterizations of P2P storage systems such as Dhash, Dhash++, PAST, TotalRecall, and Pond.

- We demonstrate a principled way to estimate the amount of extra replication required to reduce data recovery costs due to lazy repair. We also show how lazy repair can be applicable to distributed hash table (DHT)-based storage systems.
- We show that a variant of a random replica placement such as one that avoids blacklisted nodes and replaces duplicate sites is sufficient to avoid the problems introduced by the most observed correlated failures.

The rest of the paper is organized as follows. We discuss the challenges of long-term data maintenance in Section 2. In Section 3, we discuss, at a high level, how a wide-area storage system maintains data. We describe a PlanetLab trace in Section 4. We discuss data recovery in Section 5 and data placement in Section 6. Section 7 shows parameterizing existing peer-to-peer storage systems. In Section 8, we evaluate and compare the existing peer-to-peer storage systems and evaluate effects of data recovery, placement, and redundancy type on the systems. We review related work in Section 9. Finally, we conclude in Section 10.

2 Challenges

The goal of this work is to address challenges in evaluating the efficiency of maintaining data availability on a long-term basis. In respect to data availability, wide-area maintenance bandwidth can be reduced by the following three methods: data recovery, placement, and redundancy.

2.1 Data Recovery

Data recovery is key to ensuring data availability while reducing the maintenance bandwidth. It is the first area of data maintenance that we study. The goal of a successful data recovery is to repair lost redundancy only when needed and as fast as necessary. Associated problems can arise from false positives due to transient failures and repairing redundancy too fast that other services are adversely affected. Therefore, the challenge is minimizing the cost due to data recovery given that it is not possible to differentiate permanent and transient failures.

2.2 Data Placement

The second method we study for reducing wide-area bandwidth is data placement. Data placement is the process in which nodes are selected to store data replicas. The goal of data placement is to select a set of nodes that can maintain a user or system specified target data availability while consuming minimal wide-area bandwidth. There are two categories of data placement: random and selective. Random placement is used for its simplicity. Its use is often accompanied by the assumption that each node failure is independent or has low correlation[1] with each other. If node failures are not independent or have high correlation, the end result could reduce data availability. In contrast, selective placement chooses specific nodes that satisfy some constraints (e.g. select nodes that have been previously shown to have low correlation [32, 16]). One challenge to data placement is that correlated failures, which are often conjectured, can compromise the expected minimum data availability.

2.3 Data Redundancy

The final method we study to reduce cost is the efficient use of data redundancy. Redundancy is the duplication of data in order to reduce the risk of data loss. There are two categories of data redundancy: replication/mirroring and parity/erasure codes. The limitation with replication/mirroring is that it increases the storage overhead and maintenance bandwidth without comparable increase in fault tolerance. In contrast, parity/erasure codes have a better balance between storage overhead and fault tolerance. Erasure codes provide redundancy without the overhead of strict replication[9, 14, 26, 31]. However, there are negative consequences to using erasure codes[27]. Erasure codes are more complex than replication. As a result, there is a tradeoff between CPU vs data maintenance bandwidth and complexity vs simplicity when considering the use of erasure codes or replication.

3 Wide-Area Storage System Overview

In this section we present a system model for maintaining data availability. First, we begin with a set of assumptions about the system. Second, we present a basic system model. Finally, we present the redundancy and reconstruction mechanisms.

3.1 Assumptions

We are assuming a read/write archival repository where data is autonomously maintained long-term. We further assume that the deletion rate is a significantly small fraction of the overall

[1] We do not use the statistical definition of correlation; instead, we use correlation for dependence.

read, write, and data maintenance rates. Finally, we assume that the read load is balanced across the storage nodes. As a result of these assumptions, this work focuses on data maintenance, which includes the data write and repair rates, but does not include read or deletion rates.

We assume that data is maintained on nodes in the wide area and in professionally managed sites. Sites contribute resources such as nodes and network bandwidth. A node is analogous to a commodity PC with CPU, RAM, a small number of disk drives, and a set of networking ports. Nodes collectively maintain data availability. They are self-organizing, self-maintaining, and adaptive to changes. The process of copying data replicas to new nodes to maintain data availability is triggered as nodes fail.

3.2 Basic System Model

In general, a wide-area storage system maintains a *minimum* data availability with a few mechanisms. The first mechanism is the abstract use of a directory. The directory knows for each object the location of each replica and number of total and remaining[2] replicas; therefore, it knows how to resolve object location requests and when to trigger data recovery. The directory abstraction gives the storage system location independence and allows different data placement schemes to be implemented. The directory may be a central directory or a distributed directory where responsibility is partitioned among the nodes.

The second mechanism is node monitoring that is used to determine node failure. Each node sends a heartbeat periodically to the directory and the directory uses the lack of a heartbeat (detected by a timeout) to trigger data recovery.

Finally, the third mechanism is a data recovery scheme that is defined by the following three parameters: redundancy type, threshold, and amount of extra redundancy. The redundancy type refers to the use of strict replication or different erasure coding schemes[3]. The redundancy threshold is the minimum number of replicas[4] (or fragments) required to maintain data availability. And the extra redundancy is used to delay triggering data recovery. The values of the data recovery parameters change the behavior of the data recovery process and storage system as a whole; as a result, we describe the parameters further in Section 3.3

3.3 Redundancy and Reconstruction

A storage system works by maintaining at least a threshold number of replicas. When the amount of redundancy falls below the minimum threshold, the replication level is increased back to the addition of the threshold and extra redundancy. There are seven variables of concern when maintaining data in the wide-area. The redundancy type m, where m is the required number of components necessary to reconstruct the original data item. For example, $m = 1$ for replication or $m > 1$ for erasure codes [31]. The threshold th is the minimum number of replicas required to maintain the target data availability. That is, when the number of available replicas decreases below th, new replicas are created. e is the extra replicas above the threshold. e + 1 replicas have to simultaneously be down for data recovery to be triggered. Therefore, $n = th + e$ is the total number of replicas. When data recovery is triggered, enough new replicas are created to refresh the total number back to n. Next, the storage and bandwidth cost is related to the total storage overhead factor $k = \frac{n}{m} = \frac{th+e}{m}$. The reciprocal of the storage overhead factor is the rate of encoding $r = \frac{1}{k} \leq 1$. The rate of encoding is important when using different erasure coding schemes because the rate is a measure of computational complexity. Finally, the timeout to is used to detect a missing heartbeat and determine that a node has failed. Table 1 summarizes the notations.

Parameter	e.g.	Description
m	1	Number of replicas required to read object.
th	7	Threshold number of replicas required to be available.
e	1	extra replicas.
n	8	Total number of replicas $(n = th + e)$.
k	8	Total storage overhead factor $\left(k = \frac{n}{m}\right)$.
r	$\frac{1}{8}$	Rate of encoding $\left(r = \frac{1}{k} = \frac{m}{n} \leq 1\right)$.
to	1hr	Node failure detection time.

Table 1: Notations.

Using the variable notations, Figure 1 shows the replication ($m = 1$) of one object into a wide area storage system. The minimum threshold number of replicas (th) is 7 and one extra replica, $e = 1$. The total number of replicas is $n = 7 + 1 = 8$. Data recovery is triggered when 2 replicas ($e + 1$) are simultaneously down. The storage overhead factor is $k = \frac{7+1}{1} = 8$. Finally, a node failure has been detected when a heartbeat has not been received for a period of to time (e.g. 1hr).

4 PlanetLab

Our study examines wide-area storage systems using PlanetLab [2] as the example wide-area environment. PlanetLab is one of the largest open wide-area systems. It currently includes 588 nodes, hosted by 290 sites, spanning over 25 countries in 5 continents, involving over 124 universities and companies[5]. PlanetLab nodes experience many of the correlated and transient failures expected in the wide-area such as correlated failures of co-located nodes, node failures (reboot, upgrade, melt down due to heavy load, quarantine for anomaly analysis, disk failure, power failure), system failures (upgrade, compromise), and network failures (congestion, partitions, misconfigurations). As a result, PlanetLab has been used to experiment [5, 13, 25], measure [10, 11, 34, 35], and deploy [3, 20] many wide-area storage systems.

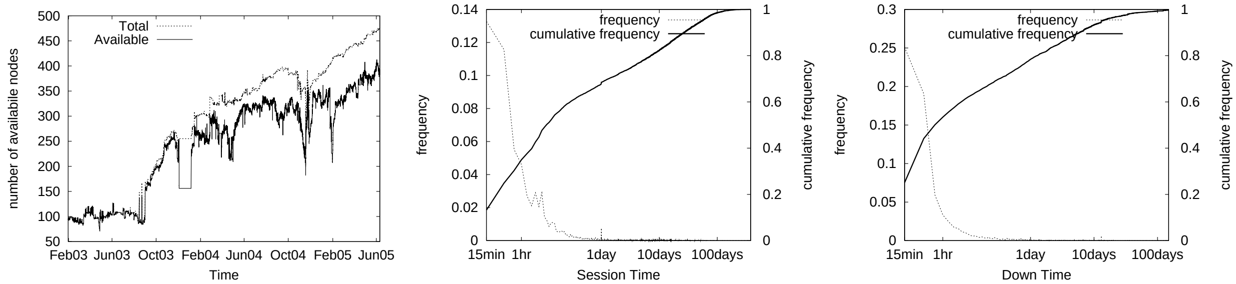
PlanetLab is one of the longest running open wide-area systems. It is currently the best suited platform to study long-term trends since it has available traces that span nearly its entire history. We expand on the work by Yalagandula et al. [34] that

[2]The number of remaining replicas is the number of replicas that reside on nodes that are currently available.

[3]Erasure codes reduce data maintenance bandwidth[9, 14, 26, 31], but their use use can be costly in terms of increased processor time and complexity; as a result, replication is often preferred.

[4]We use the term replica to refer to both a replica or a fragment.

[5]Current as of June 25, 2005



(a) Number of available nodes v Time (b) Sessiontime CDF and PDF (c) Downtime CDF and PDF

Distribution	Mean(d)	Stddev(d)	Median(m)	Mode(m)	Min(m)	90th(d)	99th(d)	Max(d)
Session	8.5	23.5	180	15	15	26.6	118.0	376.7
Down	3.5	16.9	45	15	15	5.9	73.2	388.7
Availability	0.700	0.249	0.757	1.000	0.002	0.977	1.000	1.000

(d) Sessiontime, Downtime, and Availability Statistics (d=days, m=mins)

Figure 2: PlanetLab Characterization. Details discussed in Section 4.

used three traces in their analysis, but found only PlanetLab's trace suitable for study and comparison of long-term trends.

4.1 PlanetLab Wide-area Characteristics

We used the all-pairs ping data set [30] to characterize PlanetLab. The data set was collected over a period of two years, between Feb 16, 2003 to Jun 25, 2005 and included a total of 694 nodes in that time period. We used the data set to characterize the sessiontime, downtime, and lifetime distributions. A sessiontime is one contiguous interval of time when a node is available. In contrast, a downtime is one contiguous interval of time when a node is unavailable. A node's lifetime is comprised of a number of interchanging session and down times. Sessiontime and downtime are commonly referred in the storage literature as a time-to-failure (TTF) and time-to-repair (TTR), respectively. Additionally, the average sessiontime and downtime is the mean-TTF and mean-TTR (MTTF and MTTR), respectively. The lifetime is the time between when a node first entered and last left the system (time between beginning of first session and end of last session). Availability, the percent of time that a node is up, is the total sum of the session time (sum-TTF) divided by the lifetime, which is equivalent to the more commonly known expression $\frac{MTTF}{MTTF+MTTR}$. We begin by describing how the all-pairs ping data set was collected and how we used the data set to interpret a node as being available or not.

The all-pairs ping program collects minimum, average, and maximum ping times (over 10 attempts) between all pairs of nodes in PlanetLab. Measurements were taken and collected approximately every 15 minutes from each node. The 15 minute ping period does not contain enough fidelity to detect transient failures less than 15 minutes. Measurements were taken locally from individual nodes' perspective, stored locally, and periodically archived at a central location. Failed ping attempts were also recorded.

We used a single successful ping to determine a node available. This single ping method of determining node availability was used by Chun and Vahdat [10] since a single nonfaulty path is sufficient for many routing algorithms to allow nodes to communicate.

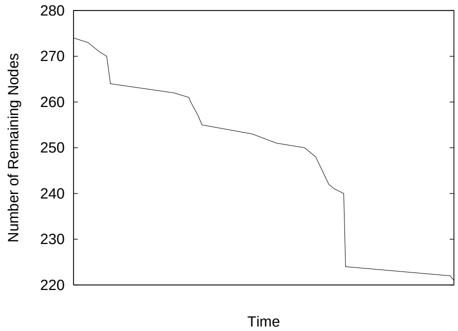
Figure 2 shows the PlanetLab characteristics using the all-pairs ping data set. Figure 2(a) shows the (total and available) number of nodes vs time. It demonstrates how the network has grown over time. More importantly, it pictorially shows the number of nodes that we simulated at each time instance in our trace.

Figure 2(b) and (c) show the frequency and cumulative frequency for the sessiontime and downtime, respectively. Note that the frequency uses the left y-axis and cumulative frequency uses the right. The mean sessiontime was 204.4 hours (8.5 days) and mean downtime was 82.8 hours (3.5 days). Both the sessiontime and downtime distributions were long tailed and the median times were 3 hours and 0.75 hours, respectively.

The sessiontimes decreased dramatically between October 2004 and March 2005 due to multiple kernel bugs that caused chronic reboot of PlanetLab nodes (shown in Figure 2(a)). The chronic reboots within the last six months of the trace doubled the total number of sessions with mostly short session times. In particular, the median session time decreased from 55.8 hours (2.3 days) between February 2003 and October 2004 to 3 hours between February 2003 and June 2005. Despite the decrease in session times, we continue to use PlanetLab as an example wide-area system to show how storage systems should adapt to changes over time without loss of data availability or increase in communication costs.

Table 2(d) summarize the sessiontime, downtime, and node availability statistics. 50% of the nodes have an availability of 75.7% or higher; however, 22% of the nodes are available less than 50% of the time.

The all-pair ping data set included more than two-years of data; however, we could not measure the node lifetime directly since many of the node names (IP addresses) lasted the entire time. As a result, we used a technique described by Bolosky et al. [7] to estimate the expected node lifetime based on node attrition. In particular, if nodes have deterministic lifetimes,



(a) Node Attrition

UpperBound	LowerBound
951 days	663 days

(b) Expected Lifetime Estimates

Figure 3: Node Attrition. Details discussed in Section 4.

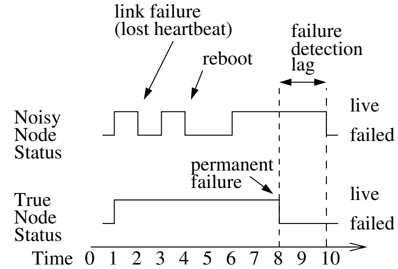


Figure 4: Transient and Permanent Failures over Time.

then the rate of attrition is constant, and the count of remaining nodes decays linearly. The expected node lifetime (meaning the lifetime of the nodes IP address, not physical hardware) is the time until this count reaches zero [7]. We counted the number of remaining nodes that started before December 5, 2003 and permanently failed before July 1, 2004. The expected node lifetime of a PlanetLab node is 951 days (Table 3(b)). Figure 3(a) shows the node attrition.

4.2 Issues and Limitations

All-pairs ping does not measure permanent failure. Instead, it measures the availability of a node name (i.e. availability of an IP address) and not the availability of data on a node. As a result, all-pairs ping was used to produce an estimated upper bound on node lifetimes. In addition, we computed an estimated lower bound on the availability of data on a node by supplementing the trace with a disk failure distribution obtained from [24] (Table 3(b)). In our experiments, the expected lifetime of a node lies between the upper and lower bound.

No all-pairs ping data exist between December 17, 2003 and January 20, 2004 due to a near simultaneous compromise and upgrade of PlanetLab. In particular, Figure 2(a) shows that 150 nodes existed on December 17, 2003 and 200 existed on January 20, 2004, but no ping data was collected in between the dates above.

5 Efficient Data Recovery

Storage systems are required to trigger data recovery and replace lost replicas as nodes fail to maintain target data availability levels. One of the problems with wide-area storage systems is differentiating permanent failures (data is lost) from transient failures (node returns with data intact). Figure 4 shows an example of transient and permanent failures over time. Transient failures that render a node temporarily unavailable are due to node reboot, system maintenance, Internet path outage, etc. In addition to transient failures, failure detection has a lag; a permanently failed node is classified as alive during the lag period. A study found that transient failures occur often in the wide-area [10].

Triggering data recovery due to transient failures can increase the cost of maintaining data severely. Some environments are very reliable and do not have much transient failures (e.g. within a data center). In contrast, other environments do not support durable storage [6] due to too many permanent and transient failures (e.g. Kazaa, Gnutella, and other high client churn environments). But for many wide-area systems, like PlanetLab [2], reliable storage can be supported and transient failures are common[10].

The cost of maintaining data is determined by the cost due to permanent failure, transient failure, write, and monitoring rate.

$$\begin{aligned} \text{Work} &= f(\text{permanent failure}, \text{transient failure}, \text{write}, \text{monitoring}) \\ &= f(\text{permanent failure}) + f(\text{transient failure}) + \\ &\quad f(\text{write}) + f(\text{monitoring}) \end{aligned} \tag{1}$$

Although the overall cost is additive, the cost due to transient failures often dominates. Intuitively, the overall data maintenance cost is decreased by decreasing the cost due to transient failures. We decrease the cost due to transient failures by adding extra replicas. Decreasing the reaction to transient failure is analogous to decreasing error rate of sending a message across a noisy channel by adding extra bits. Explicitly, we tradeoff increased storage for reduced communication while maintaining the same minimum data availability. The extra replicas absorb noise. This translates into a decreased rate of triggering data recovery since it is less likely for the extra replicas to simultaneously be down. Figure 5 illustrates the breakdown in the data maintenance costs as the extra replicas are increased.

Figure 5 is key. It shows that with no extra replicas the probability of triggering data recovery due to transient failures is actually quite high; hence, the cost due to transient failures is high. In fact, the probability of triggering data recovery is *higher* than the probability of a single timeout since the chance of *any* one out of n replicas timing out is higher than the probability of a single timeout. If we add extra replicas and require that at least *all* extra replicas simultaneously be down, then the rate of triggering data recovery drops significantly; similarly, the cost due to transient failures drops significantly.

The goal is to estimate the optimal number of extra replicas required to minimize work. In the following subsections we describe data recovery optimizations. In particular, we dis-

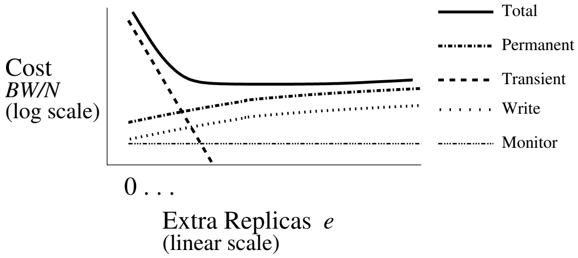


Figure 5: Example. Cost per Node of Maintaining Data in a Wide Area Storage System.

Variable	Description
$1 - \epsilon$	Target minimum data availability.
ϵ	Probability data is *un*available.
a	Average node availability.
D	Total amount of unique data.
S	Total amount of storage ($S = kD$).
N	Total number of nodes.
T	The average lifetime of all N nodes.
$u(x)$	Probability distribution function of downtimes
to	Timeout used to determine a node is unavailable.
to_{max}	Maximum time a node has been unavailable and came back.
p_{dr}	Rate of triggering data recovery
p_{to}	Probability node down longer than timeout to.

Table 2: Notation

cuss how to set the minimum data availability threshold and estimate the number of extra replicas that minimize work.

5.1 Estimator Algorithm

In this section, we show how to estimate the amount of extra replicas required to absorb "noise" and reduce rate of triggering data recovery. The algorithm works as follows. Given a target data availability to maintain, first it calculates the minimum threshold number of replicas required (calculation based on average node availability). It then supplements this number with a set of extra replicas to absorb noise (calculation based on average node lifetime, sessiontime, and downtime). Finally, it triggers data recovery when *all* extra replicas are simultaneously considered down. In the following subsections we first show how to estimate the minimum data availability threshold, then we show how to estimate the amount of extra replicas.

5.1.1 Estimating the Minimum Data Availability Threshold

Bhagwan et al. [4] and Blake and Rodrigues [6] demonstrated how to calculate the minimum data availability threshold for both replication and erasure-resilient data redundancy schemes. For instance, the minimum data availability for a replication scheme ($m = 1$) equals 1-p[no replicas available] or $1 - \epsilon = 1 - (1 - a)^{th}$, where a is the average node availability, th is the minimum data availability threshold, and ϵ is the probability that no replicas are available. As a result, $th = \lceil \frac{\log \epsilon}{\log (1-a)} \rceil$. The equality assumes that all nodes are independent. It also assumes a replication redundancy scheme[6].

5.1.2 Estimating the Amount of Extra Replicas

We estimate the optimal number of extra replicas e by synthesizing the cost due to data maintenance as expressed in Equation 1. In particular, we develop an estimator for each term in Equation 1, then calculate each term's cost, and pick e where the overall cost is minimal. The key is to pick the optimal number of extra replicas e that reduces the cost due to transient failures without increasing the cost due to writes or permanent failures too high. We discuss each terms' estimator and the overall data maintenance estimator in turn below.

[6]The minimum data availability threshold can similarly be calculated for an erasure coded scheme using equations described by Bhagwan et al. [4] and Blake and Rodrigues [6]

Permanent Failure Estimator Permanent failure is the loss of data on a node. The data maintenance cost due to permanent failures is dependent on the average amount of storage per node $\frac{S}{N}$ and the average storage node lifetime T.

$$\text{permanent}\frac{BW}{N} = O\left(\frac{S}{NT}\right) \quad (2)$$

where total storage S is the product of the total amount of unique data D and storage overhead factor k (i.e. $S = kD$). Recall from Section 3.3 that $k = \frac{th+e}{m} = \frac{n}{m}$. Equation 2 states that on average the total storage S must be copied to new storage nodes every average node lifetime T period. We assume that all storage nodes have a finite lifetime (e.g. 1-3 years) typical of a commodity node, so storage will not be biased towards one ultra reliable node. Equation 2 has been discussed in literature by Blake and Rodrigues [6].

Transient Failure Estimator Transient failure is when a node returns from failure with data intact. Reducing the rate of triggering data recovery due to transient failures reduces the amount of unnecessary data recovery. We assume that a timeout to is used to determine if a node has failed or not. p_{to} is the probability of a single timeout. If there is no extra replicas, then the probability of *at least* one node timing out is high; as a result, the rate of triggering data recovery, p_{dr}, is high. That is,

$$\begin{aligned} p_{to} &= P(\text{storage down longer than } to) \\ &= \int_{to}^{\infty} u(x)dx \quad (3) \\ p_{dr} &= P(\text{at least one storage node down longer than } to) \\ &= \sum_{i=1}^{n=th} \binom{n}{i} p_{to}^{i}(1-p_{to})^{n-i} \quad (4) \end{aligned}$$

where $u(x)$ is the probability distribution function of down times. Figure 2(c) is the downtime distribution for PlanetLab. Equation 4 is the probability that at least one of the n replicas times out. Note that the probability of at least one of n replicas timing out is higher than the probability of a single time out p_{to}. This assumes that failures are independent.

Lets now assume we add extra replicas to the minimum data availability threshold. We require that at least *all* extra replicas to simultaneously be down in order to trigger data recovery. As a result, the rate of triggering data recovery is

$$
\begin{aligned}
p_{dr} &= P(\text{at least all } \textit{extra} \text{ replicas down longer than } to) \\
&= \sum_{i=e+1}^{n=th+e} \binom{n}{i} p_{to}^{i} (1 - p_{to})^{n-i}
\end{aligned} \tag{5}
$$

Equation 5 computes the probability that at least $e + 1$ nodes have simultaneously timed out. It also shows that the rate of triggering data recovery can be reduced by increasing the extra replicas. For example, given a timeout period $to = 1$ hour and a probability of a timeout $p_{to} = 0.25$, then for the following parameterization $m = 1, th = 5, e = 4 (n = 9 = 5 + 4)$, the resulting rate of triggering data recovery is $p_{dr} = 0.049$, which is significantly less than the probability of triggering data recovery with no extra replicas $p_{dr} = 0.762$ ($m = 1, th = 5, n = 5, e = 0$).

The cost of triggering data recovery is the amount of storage per node $\frac{S}{N}$ and the average period for the MTTF and MTTR (i.e. average session and downtime). The transient term is

$$
\text{transient}\frac{BW}{N} = p_{dr} \cdot O\left(\frac{S}{N(MTTF + MTTR)}\right) \tag{6}
$$

Write Rate Estimator The write rate is the rate of unique data being added to the storage system. The cost due to writes is simply the unique write rate multiplied by the storage overhead factor k.

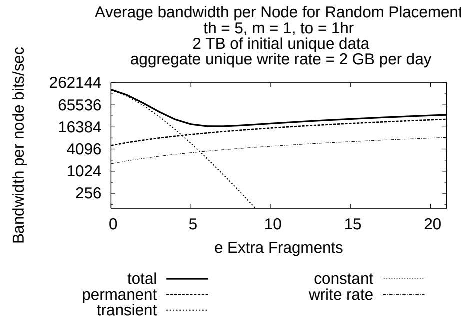
$$
\text{write}\frac{BW}{N} = k \cdot \text{write rate} \tag{7}
$$

Ideally, we want $k = \frac{th+e}{m} = \frac{n}{m}$ to be small. However, the cost due to writes may be increased until an optimal number of extra replicas e is obtained since k depends on e.

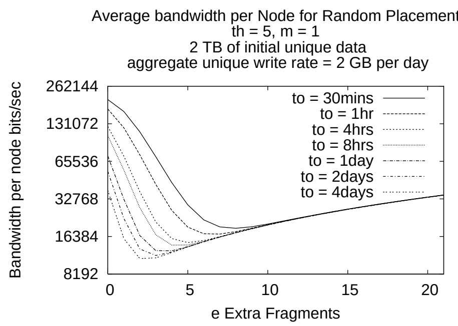
Heartbeat Estimator A heartbeat is used to determine whether a node is alive or not. We assume that each node serves as a directory node and can trigger data recovery as described in Section 3.2. As a result, each node needs to know the status of all other nodes. The cost per node for monitoring all other nodes is dependent on the number of nodes N, the heartbeat timeout period to, and the size of a heartbeat hb_{sz}, That is,

$$
\text{heartbeat}\frac{BW}{N} = \frac{N}{to} \cdot hb_{sz} \tag{8}
$$

Equation 8 states that each node sends a heartbeat to all other nodes every to period. For most reasonable timeouts, the cost due to heartbeats will not be a significant fraction of the overall data maintenance costs. For example, if $N = 10000$ nodes, $to = 1$ hour, and $hb_{sz} = 100$ B, then heartbeat$\frac{BW}{N} = 277.8$ Bps.



(a) $to = 1$ hour



(b) Vary to

Figure 6: Data Maintenance Estimator for Storage Systems on PlanetLab.

5.1.3 Example of Applying Data Maintenance Estimator

Figure 6 shows an example of applying the data maintenance estimator to maintain four 9's of data availability (i.e. 1 out of every 10,000 attempts to access an object will fail). We assume an aggregate amount of unique data is $D = 2TB$, the aggregate unique write rate is $2GB$ per day, the number of nodes is $N = 400$, and the timeout value is $to = 1$ hour. Finally, we use the expected node availability, lifetime, MTTF, and MTTR from Figure 2(d) and the downtime distribution from Figure 2(c). Using the node availability, we calculated the minimum data availability threshold to be $th = 5$. Figure 6(a) shows that the estimated optimal number of extra replicas that minimizes the cost due to data maintenance is six. Similarly, Figure 6(b) shows the estimated optimal number of extra replicas for varying number of timeout values.

Notice that data maintenance estimator computation can be performed locally at a each node with local estimates for the total number of storage nodes, average node availability, lifetime, MTTF, MTTR, storage per node, and write rate per node. This is beneficial because the data maintenance estimator can be performed online so that the storage system parameterization can adapt to the changing environment characteristics overtime.

6 Data Placement Strategies

The second component of data maintenance that we study is data placement. Data placement is the process in which nodes are selected to store data replicas. The goal of data placement is to select a set of nodes that can maintain a user or system specified target data availability while consuming minimal wide-area bandwidth. It includes selecting nodes that are likely to be available, not correlated[7] when failures happen, and spread load.

There are two categories of data placement: random and selective. Random placement is used for its simplicity. Its use is often accompanied by the assumption that each node failure is independent or has low correlation with each other. If node failures are not independent or have high correlation, the end result could reduce data availability or increase data maintenance cost to maintain the same data availability. In contrast, selective placement chooses specific nodes that satisfy some constraints (e.g. select nodes that have been previously shown to have low correlation [1, 15, 16, 19, 32]).

Many existing storage systems on PlanetLab use a random placement strategy; however, many papers in literature have cited a long-tailed distribution for node downtimes and significant correlated downtimes for PlanetLab nodes [10, 23, 34]. Random is the choice data placement strategy for many storage systems on PlanetLab because it is a simple algorithm and most PlanetLab nodes are reliable. Additionally, Random spreads replicas uniformly across the storage nodes. However, Random is likely to place replicas on "flaky" nodes as well as the reliable ones. Moreover, Random is likely to place multiple replicas in duplicate sites. For example, nearly 22% of PlanetLab nodes are in Berkeley, so if more than 10 replicas are used then it is likely more than one will be placed in Berkeley.

Unreliable and correlated PlanetLab nodes have been sited in literature. However, it is not clear to what degree "flaky" and correlated nodes affect the cost of data maintenance. We compare the data maintenance cost of random placement strategies that blacklist flaky nodes and/or avoid placing multiple replicas in duplicate sites. In particular, we blacklist nodes that are available less than 50% of the time; 10% of PlanetLab nodes are available less than 50% of the time according to Figure 2(d)). Additionally, to avoid placing multiple replicas in duplicate sites, we pick another random node to store a replica if a node in a duplicate site was already selected.

The four variations of random data placement strategies that we compare are `Random`, `RandomBlacklist`, `RandomSite`, and `RandomSiteBlacklist`. `Random` placement picks n unique nodes at random to store replicas. `RandomBlacklist` placement is the same as `Random` but avoids the use of nodes that show long downtimes. The blacklist is comprised of the top z nodes with the longest total downtimes. `RandomSite` avoids placing multiple replicas in the same site. `RandomSite` picks n unique nodes at random and avoids using nodes in the same site. We identify a site by the 2B IP address prefix. The other criteria can be geography or administrative domains. Finally, `RandomSiteBlacklist` placement is the combination of `RandomSite` and `RandomBlacklist`.

The other category of data placement is selective data placement. We can set up an optimization problem to minimize maintenance bandwidth under data availability constraints. However, this problem is NP-hard. Therefore, we use an offline algorithm based on heuristics. The benefits of these more sophisticated selective placement strategies are not well understood in terms of the cost of data maintenance (i.e. reduced triggering of data recovery; hence, reduced data maintenance costs). We compare costs of the random placement strategies discussed above against the best offline algorithm, among the ones we studied, that uses future knowledge when selecting nodes for data placement.

The offline clairvoyant selective data placement strategy, named `Max-Sum-Session`, uses future knowledge of node lifetimes, sessiontimes, and availability to place replicas. In particular `Max-Sum-Session` places replicas on nodes with the highest remaining sum of session times. This strategy places replicas on nodes that permanently fail furthest in the future and exhibit the highest availability.

7 Parameterizing Existing Storage Systems

Using the parameters described in Section 3.3, we show (in Section 7.2) the parameterization of five existing storage systems: Dhash [12], Dhash++ [13], PAST [17], Pond [25], and TotalRecall [5]. We describe these storage systems since they have been deployed and measured on PlanetLab, their parameterizations are described in literature, and many other notable storage systems are derived from them.

The existing storage systems that we parameterized fall into two categories: distributed hash table (DHT) and directory (DHT+level of indirection). Dhash, Dhash++, and PAST are DHT-based storage systems and Pond and Total Recall are directory based-storage systems. A DHT-based storage system differs from the directory-based storage system that we described in Section 3.2; we compare and contrast the differences in Section 7.1.

In Section 8, we demonstrate the cost of maintaining data long-term on PlanetLab using these storage system parameterizations.

7.1 DHT- and Directory-based Storage Systems

There are quite a few differences between DHT- and directory-based storage systems. The differences include flexibility for placement algorithms, triggering of data recovery, object size constraints, and the number of nodes monitored.

First, a DHT-based placement algorithm is random but restrictive; whereas, a directory-based storage system is flexible to implement many different placement algorithms due to the level of indirection. The difference in placement algorithms available to each storage system is based on how the storage system works.

A DHT-based storage sytem works by consistently hashing an identifier space over a set of nodes. An identifier is assigned to each node using a secure hash function like SHA-1. Similarly, using the same secure hash function, an identifier is

[7] We use correlation to mean dependence, not the the statistical definition of correlation.

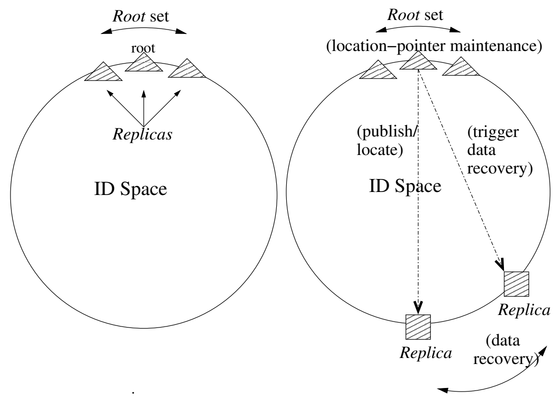


Figure 7: DHT- and Directory- Based Storage System Architectures.

assigned to each object. Both node and object identifiers are drawn from the same name space. The node that is responsible for a range of identifier space is the *root* of the identifier range. For example, in PAST [17], the node whose identifier is numerically closest to the object identifier (in the identifier ring), is responsible for the object; whereas, in Dhash [12] the node whose identifier immediately follows the object identifier is the root. A DHT prevents data loss when the root node is unavailable by defining a *root set*, redundant set of root nodes that mirror each other. For example, both Chord successor-list [29] and Pastry leafset [28] are root sets. To be explicit, each unique identifier has a unique root and each root has a unique root set (root set$_i \not\subset$ root set$_j$, $\forall i \neq j$). All root sets are the same size, and the size of a root set is smaller than the size of the network. Since each object has a unique root, an object is replicated to a well-defined root set. Figure 7(a) pictorially shows how a root, root set, and replicas for a particular object are organized around the identifier ring.

Unlike DHTs, directory-based storage systems such as Total Recall and Pond use a level of indirection to place, locate, and maintain data. A level of indirection allows more flexibility in data placement. Total Recall and Pond implement the directory abstraction described in Section 3.2, except that they use a DHT to store location-pointers. To be precise, they store replicas for a particular object on different storage nodes and use a DHT to store and maintain replica location-pointers. Namely, each replica (for a particular object) has the same object identifier[8], but a different location. As a result, a particular object has a unique root node that resolves replica location requests and triggers data recovery to repair lost replicas. Figure 7(b) shows a directory system architecture. To trigger data recovery, the root node needs to know the status of all the storage nodes for a particular object. When a storage node is declared down, data recovery is triggered.

Second, a directory-based storage system only triggers data recovery as nodes fail; whereas, a DHT-based storage system potentially triggers data recovery on node joins and failures.

[8]Each fragment for a particular object has the same identifier if we use erasure codes [33].

Parameter	Dhash	Dhash++	PAST	Pond	TotalRecall
m	1	7	1	16	15
th	5	14	9	24	variable
e	0	0	0	6	10
n	5	14	9	32	$th + 10$
k	5	2	9	2	≈ 3
r	N/A	$\frac{1}{2}$	N/A	$\frac{1}{2}$	$\approx \frac{1}{3}$
to (min)	< 15	< 15	< 15	< 15	< 15
Placement	DHT-Random	DHT-Random	DHT-Random	Random	Random

Table 3: Existing Storage System Parameterization

That is, a DHT continually rebalances data when a node is added to or removed from the system; while, a directory-based storage system does not rebalance data. In fact, if the data placement strategy for a directory is random, then the amount of data a storage node has is proportional to the node's current lifetime in the system.

Third, if using a directory-based storage system, then we are assuming that the storage due to location-pointers is significantly less than the storage due to data and that the total number of replicas of all objects is greater than the number of nodes. If either assumption is violated then a directory storage strategy should not be used.

Finally, DHT and directory based storage systems differ in the number of nodes they monitor. A DHT has to monitor only $O(n)$ nodes (i.e. n is the size of the root set and total number of replicas), where as, a directory has to monitor all N storage nodes. A directory-based storage system can still be efficient because the number of replicas per node is assumed to be greater than the number of nodes.

7.2 Existing Storage Systems

Dhash Dhash is a replicated DHT-based storage system. It replicates each object onto five storage nodes. It does not define any extra replicas, so it immediately creates a new replica as soon as a failure has been detected or a new node joins the root set. This has been described as eager repair [5]. The placement scheme for Dhash is DHT-Random. A DHT-Random placement is a random placement scheme since node identifiers are randomly distributed in a DHT; however, the node responsible to store a given data item is deterministic given the node and object identifier. As a result, a DHT-Random placement scheme is more restrictive than a purely random placement scheme. Finally, Dhash uses a fairly aggressive timeout value. This is due to the tight coupling between the storage system and the networking overlay.

Dhash++ Dhash++ is the same as Dhash except that Dhash++ uses erasure codes instead of replication. Dhash++ uses a rate $r = \frac{1}{2}$ erasure code with $n = 14$ total fragments and $m = 7$ fragments required to reconstruct the object. Dhash++ eagerly repairs lost redundancy since it does not define any extra replicas.

PAST PAST parameterization is the same as Dhash except that it uses more replicas. PAST maintains $logN$ replicas, so stores 9 replicas for a system of a 512 storage nodes.

Pond Pond is a directory-based storage system and uses a level of indirection to place, locate, and maintain data. For long-term data maintenance of large objects, Pond uses erasure codes. Pond uses a rate $r = \frac{1}{2}$ erasure code with $n = 32$ total fragments and $m = 16$ fragments required to reconstruct an object. Pond defines an arbitrary threshold that is half of $n - m$, so $th = 24$ and $e = 6$. The threshold used in Pond was retrieved from the code available at http://oceanstore.sourceforge.net.

Total Recall Similar to Pond, Total Recall is a directory-based storage system and uses erasure codes. However, unlike all of the above storage systems, Total Recall allows the target data availability to be specified on a per object bases. Furthermore, Total Recall dynamically maintains a minimum data availability. That is, the the minimum data availability threshold th fluctuates dynamically as the node availability fluctuates.

Table 3 summarizes the parameterization of the above existing storage systems.

8 Evaluation

In this section, we present a detailed performance analysis via a trace driven simulation of maintaining data on PlanetLab. Our evaluation illustrates the data maintenance characteristics of existing storage systems and highlights promising ways to estimate and tune their performance. In particular, we show that many existing storage systems make arbitrary decisions that are costly. For instance, we show the estimated number of extra replicas that yield an order of magnitude reduction in the amount of bandwidth that Dhash [12] uses to maintain 2TB of data on PlanetLab.

First, we present our evaluation methodology in Section 8.1. In Section 8.2, we evaluate existing storage systems. We evaluate data recovery optimizations in Section 8.3. In Section 8.4, we compare data placement strategies. Finally, in Section 8.5 we compare different data redundancy schemes.

8.1 Evaluation Methodology

Our evaluation methodology is discussed in the following order: trace used, interaction between trace and simulator, amount of data maintained, target data availability maintained, node failure detection to maintain data, and cost metrics.

We used the PlanetLab trace supplemented with a disk failure distribution as described in Section 4. The trace-driven simulation ran the entire two-year trace of PlanetLab (Figure 2(a)). The simulator added a node at the time a node was available in the trace and removed it when it was not available. Nodes that were not available in the trace were not available in the simulator (and visa versa).

We simulated maintaining 2TB of initial unique data comprised of 32k objects each of size 64MB. Since 64MB is the same size as a GFS extent [18], we used it as our unit of maintenance. Additionally, the storage overhead factor, $k = \frac{n}{m}$, increased the size of the repository. For example, with $n = 10$ and $m = 1$ the total size of the repository was now 320k replicas and 20TB of redundant data.

In addition to the initial data, we continuously added new unique data to the repository increasing its size over time. We used two different write rates to add new data, 10Kbps and 1Kbps per node (e.g. 20GB and 2GB per day with 200 nodes). To make the write rates concrete, we used the measurement that an average workstation creates 35MB/hr (or 10Kbps) [7] of total data. Most of the writes were temporary, so we used a permanent write rate of 3.5MB/hr (or 1Kbps). Finally, 10Kbps and 1Kbps per node correspond to 20GB and 2GB of unique data per node per year added to the repository, respectively. With an average of 200 nodes, the system would increased in size at a rate of 20TB and 2TB per year, respectively.

Unless otherwise noted, in most of our experiments we used replication and maintained a minimum threshold of five replicas ($th = 5$). We arrived at this minimum threshold by setting the target data availability to four 9's as suggested by Rodrigues and Liskov [27] and using equations described in Section 5.1.2.

We maintained data by implementing timeouts/heartbeats in the simulator to detect node failures and trigger data recovery as a result of failed extra replicas. In both directory- and DHT-based storage systems, heartbeats were sent from an individual node's perspective at the same period as the timeout to. A node in a directory-based storage system sent a heartbeat to all other nodes. In contrast, a node in a DHT-based storage system sent a heartbeat to only its root set neighbors (Figure 7(a)). The timeout used to detect failure was $to = 1$hr for most of the experiments, which was picked as a sufficient value after we experimented with a variety of timeout values ranging from 15 minutes to 4 days. Data recovery was triggered when the extra replicas plus one, $e+1$, were simultaneously considered down (description in Section 3).

The simulator measured the total number of repairs triggered vs time and the bandwidth used per node vs time. Additionally, we measured the average bandwidth per node as we vary the timeout and extra replicas. Finally, we measured the average number of replicas available vs time. But we omit this graph since the number of replicas was actually at least the threshold.

8.2 Evaluation of Existing Storage Systems

In this section, we measure and compare the cost that existing storage systems incur to maintain 2TB of initial data and a daily write rate of 2GB. Further, we show the reduction in cost after tuning each system. Each system maintains data for two years on PlanetLab.

The comparison was conducted as follows. First, we parameterized each storage system with the configurations described in literature, as shown in Table 3. Second, we ran each configuration through the trace-driven simulation of PlanetLab and measured their associated costs. Third, using the same parameters, we added an estimated number of extra replicas to each system. Finally, we ran the tuned configuration through the simulation and measured the associated costs. Table 4 shows

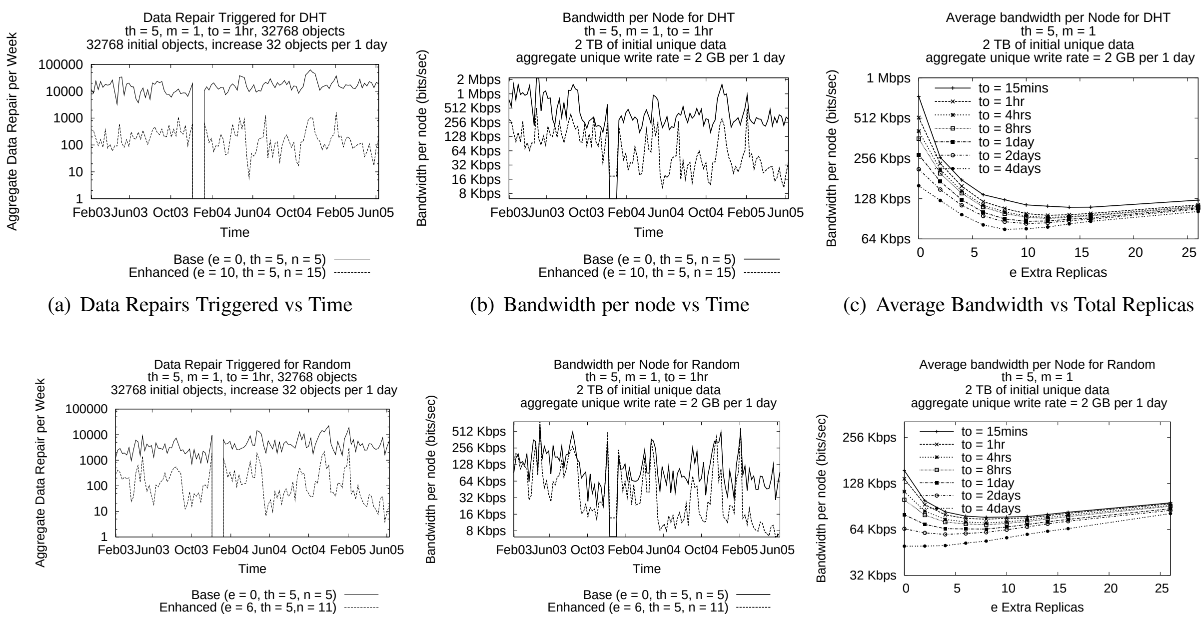
	Dhash ($m=1$,$th=5$,e=**0**)	**Dhash++** ($m=7$,$th=14$,e=**0**)	**PAST** ($m=1$,$th=9$,e=**0**)	**Pond** ($m=16$,$th=24$,e=**6**)	**TotalRecall** ($m=15$,$th=29$,e=**10**)
$1-\epsilon$ (in # 9's)	4	3	7	2	4
# Repairs	14,431,970	10,550,003	24,701,667	1,041,089	1,312,824
BW/N (Kbps)	516.7	409.0	865.1	61.1	77.6

(a) Before Applying Estimator

	Dhash ($m=1$,$th=5$,e=**6**)	**Dhash++** ($m=7$,$th=14$,e=**12**)	**PAST** ($m=1$,$th=9$,e=**6**)	**Pond** ($m=16$,$th=24$,e=**16**)	**TotalRecall** ($m=15$,$th=29$,e=**16**)
$1-\epsilon$ (in # 9's)	4	3	7	2	4
# Repairs	447,204	564,249	984,951	339,727	510,514
BW/N (Kbps)	75.5	52.2	262.3	33.5	45.6

(b) After Applying Estimator

Table 4: Cost of Maintaining 2TB of unique data for two years on PlanetLab using Existing Storage System Parameterizations. (a) costs incurred with parameterizations as described in literature. (b) costs incurred after applying estimator. The data availability $(1-\epsilon)$, total number of repairs triggered, and average bandwidth per node are highlighted in the first, second, and third rows, respectively. Note that the data availability is based on the parameters m and th. to =1hr for each system.



(a) Data Repairs Triggered vs Time (b) Bandwidth per node vs Time (c) Average Bandwidth vs Total Replicas

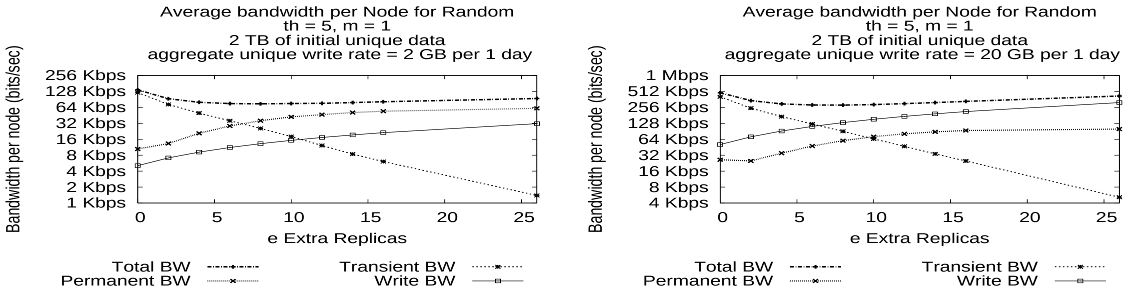
(d) Data Repairs Triggered vs Time (e) Bandwidth per node vs Time (f) Average Bandwidth vs Total Replicas

Figure 8: Data Recovery Optimizations. Figures (a), (b), and (c) use the DHT-based storage system like Dhash and Figures (d), (e), and (f) use a directory-based storage system with a Random placement. Figures (a) and (d) shows the number of repairs triggered per week over the course of the trace. Figures (b) and (e) show the average bandwidth per node (averaged over a week) over the course of the trace. Finally, Figures (c) and (f) show the average bandwidth per node as we vary the number of extra replicas and timeout values.

the resulting costs in terms of number of repairs triggered and average maintenance bandwidth per node.

Table 4(a) illustrates that most existing storage systems made arbitrary configuration decisions that were costly. In particular, the effective target data availability was often either too high or too low; Dhash++ and Pond maintained three and two 9's, respectively, and PAST maintained seven 9's. Furthermore, the number of repairs triggered and associated bandwidth per node costs were excessive. For example, Dhash and its successor Dhash++ used nearly 0.5Mbps for background maintenance bandwidth. On the other hand, systems tuned for data maintenance, like TotalRecall, used less resources (77Kbps); however, still made some unnecessary arbitrary decisions like $e = 10$ extra fragments.

Applying the data maintenance estimator improved the performance of each system. Table 4(b) shows the estimated number of extra replicas for each system. It also demonstrates significant cost reductions to maintain the same data availability as the untuned systems. For instance, for Dhash, the number of repairs triggered reduced two-fold from 14M to 447k and data maintenance bandwidth reduced from 0.5Mbps to 75Kbps. Similar decreases can be seen in Dhash++ and PAST. Pond and TotalRecall benefited from the data maintenance estimator as well, but gains were not as pronounced since the



(a) Cost Breakdown vs Number Replicas (1Kbps) (b) Cost Breakdown vs Number Replicas (10Kbps)

Figure 9: Cost Breakdown for Maintaining Minimum Data Availability for 2 TB of unique data. (a) and (b) Cost breakdown with a unique write rate of 1Kbps and 10 Kbps per node, respectively. Both (a) and (b) fix the data placement strategy to `Random` and timeout $to = 1$hr. The cost due to heartbeats is not shown since it was less than 1Kbps.

systems parameters have already been tuned for data maintenance. However, the data maintenance estimator removes much of the ambiguity even for TotalRecall and Pond.

In addition to static configurations, the data maintenance estimator can be applied online as described in Section 5.1.3. For instance, we modeled TotalRecall where the target data availability was four 9's (same as above) and threshold and extra replicas were dynamically maintained according to the current data maintenance estimator. This online data maintenance estimator further reduced the number of repairs triggered to 266,518

8.3 Evaluation of Data Recovery Optimizations

Our second analysis evaluates the use of data recovery optimizations. For this analysis, we maintain four 9's of data availability using a DHT- and directory-based storage system with a timeout value of $to = 1$hr. In Figure 8, we measure the number of repairs triggered and average bandwidth per node over time for the optimal and worst number of extra replicas. Additionally, we show the breakdown in cost in Figure 9. Note that the DHT-based storage system parameterization is the same as Dhash (i.e. $m = 1$ and $th = 5$) and the directory-based data placement strategy is `Random`.

The results in Figure 8 show that in both the DHT- and directory-based storage system (Figures a-c and d-f, respectively), the configurations that use the estimated optimal number of extra replicas use at least an order of magnitude less bandwidth per node than with no extra replicas. More importantly, Figures (c) and (f) show that large timeout values exponentially decrease the cost of data maintenance; however, the increase in node failure detection potentially compromises data availability. An alternative solution was a linear increase in extra replicas which similarly exponentially decreased the cost of data maintenance without sacrificing data availability. Figures (c) and (f) are consistent with the expected costs as depicted in Figure 6(b).

Figure 9 showed the breakdown in bandwidth cost for maintaining a minimum data availability threshold and extra replicas. Figure 9 fixed both the timeout $to = 1$hr and data placement strategy to `Random`. Figure 9(a) and (b) used a per node unique write rate of 1Kbps and 10Kbps, respectively. Both Figures 9(a) and (b) illustrated that the cost of maintaining data due to transient failures dominated the total cost. The total cost was dominated by unnecessary work. As the number of extra replicas, which are required to be simultaneously down in order to trigger data recovery, increased, the cost due to transient failures decreased. Thus, the cost due to actual permanent failures, which was a system fundamental characteristic, dominated. The difference between Figure 9(a) and (b) is that the cost due to permanent failures dominated in (a) and the cost due to new writes dominated in (b). Finally, the cost due to sending heartbeats to each node in an all-pairs ping fashion once an hour was insignificant. These results are consistent with the data maintenance estimator depicted in Figure 6(a).

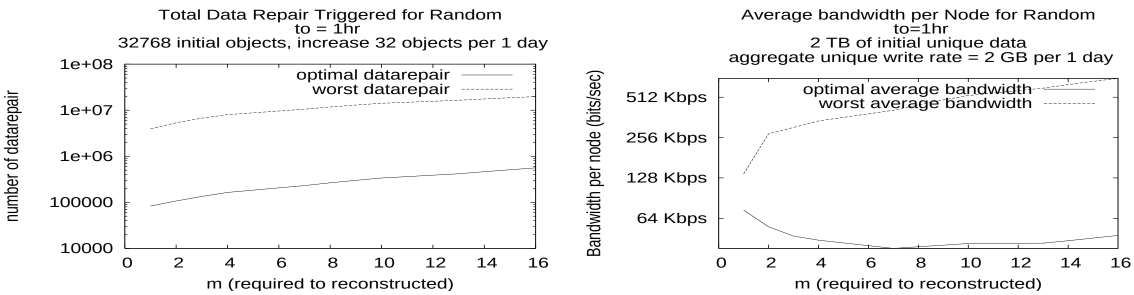
8.4 Evaluation of Data Placement Strategies

Our third evaluation compares data placement strategies. We first compare the different random, DHT, and clairvoyant data placement strategies. Table 5 shows for all the placement strategies the total number of repairs triggered, average bandwidth per node, and percentage of improvement over `Random`. Additionally, Table 5 the average and standard deviation of the number of replicas per node. The storage system parameters were replication redundancy scheme $m = 1$, minimum threshold number of replicas $th = 5$, and a heartbeat timeout $to =$ 1hr. The size of the blacklist for the `RandomBlackList` and `RandomSiteBlacklist` placement strategies was the top 35 nodes with the longest total downtimes. Table 5 shows how the data placement strategies differ in cost for the estimated optimal number of extra replicas $e = 6$; thus, the total number of replicas per object was $n = 11$.

Table 5 shows that more sophisticated placement strategies exhibited noticeable increase in performance; that is, fewer repairs triggered compared to `Random`. For example, the `RandomSiteBlacklist` placement showed a 4.38% improvement over `Random`, which was slightly more than the sum of parts, 1.87% and 2.47% for `RandomSite` and `RandomBlacklist`, respectively. The clairvoyant placement strategy exhibited a 7.67% improvement (`Max-Sum-Session`). The `DHT` placement consumed more

	Data Placement Strategy. ($m = 1$, $th = 5$, $n = 11$, |blacklist| $= 35$, $to = 1$hr)					
	Random	DHT	RandomSite	RandomBlacklist	RandomSiteBlacklist	Max-Sum-Session
# Repairs	227,242	447,204	223,003	221,618	217,291	209,815
% Improvement		-96.80	1.87	2.47	4.38	7.67
BW/N (Kbps)	75.5	121.8	74.5	74.2	73.2	70.5
% Improvement		-61.37	1.25	1.70	2.94	6.62
# Replicas/N	1386.2 ± 1126.6	1439.9 ± 684.4	1381.9 ± 1140.5	1391.8 ± 1182.1	1386.5 ± 1193.7	1388.9 ± 1213.3

Table 5: Comparison of Data Placement Strategies. $n = 11$ and $e = 6$.



(a) Total Data Repairs Triggered vs m

(b) Bandwidth per node vs m

Figure 10: Data Redundancy. m (the number of replicas/fragments required to reconstruct an object) vs Cost. (a) Cost–number of triggered data repairs. (b) Cost–average bandwidth per node.

bandwidth per node and triggered more data repairs than Random; however, the distribution of the number of replicas per node was more uniform for DHT as can be seen with the smaller standard deviation of 684 replicas per node.

8.5 Evaluation of Data Redundancy

Our final analysis compares the use of replication and coding data redundancy schemes. As we vary the data redundancy type m, Figure 10 shows the optimal and worst total number of repairs (Figure 10(a)) and average bandwidth per node (Figure 10(b)). Higher erasure types (e.g. $m > 1$) show similar benefits in using extra replicas. The advantage of using erasure codes is that extra replicas can be added without significantly increasing the storage overhead factor. For example, the storage overhead factor was $k = \frac{th+e}{m} = \frac{5+6}{1} = 11$ for the optimal extra replicas with $m = 1$; whereas, the storage overhead was $k = \frac{31+16}{16} \approx 3$ using erasure codes $m = 16$. However, the benefit of coding is reduced since the number of repairs triggered increases resulting in a similar average optimal bandwidth per node of approximately 64Kbps.

9 Related Work

Many wide-area storage systems use timeouts/heartbeats to detect node failures. Wide-area storage systems often use heartbeats to determine that a node is available and lack of a heartbeat to determine a node is unavailable. After a node is classified as unavailable, the system may trigger data recovery. The problem is triggering data recovery unnecessarily. In particular, It is well known that in distributed systems it is often not possible to determine the exact reason two nodes cannot communicate. For example, it is not possible for a node to distinguish a network partition (lossy link to bad router configuration), a node being transiently down, and a permanent failure (data on node removed permanently from the network). Many well known wide area storage systems trigger data recovery almost immediately after a timeout [12, 17, 20]. Some systems, however, use erasure codes and delay triggering data recovery [5, 8, 13, 21, 25].

The alternative to increasing the number of extra replicas is increasing the timeout as suggested by [27]. The problem with increasing the timeout too high is compromising the minimum data availability since the time to detect a permanent failure is high. Whereas, increasing the number of extra replicas decreases the rate of triggering data recovery without compromising the minimum data availability.

Erasure codes reduce data maintenance bandwidth[9, 14, 26, 31], but are orthogonal to extra replicas. Erasure-resilient systems were not designed to handle noisy node failure signals. Instead, erasure codes are used to lazily repair lost replicas while ensuring data availability. They are a natural fit with a minimum data availability threshold and extra replicas. However, as far as we know, all systems that use erasure codes define an arbitrary threshold; as a result, they define an arbitrary number of extra replicas. We define a principled way to tune the number of extra replicas.

There is a cost to using erasure codes and sometimes replication is desired. Erasure codes are processor intensive to produce, as a result, there is a tradeoff between CPU and networking when considering the use of erasure codes or replication. Additionally, replication is desired for simplicity. For example, most Distributed Hash Tables (DHTs) prefer simplicity and use replication.

Extra replicas do not reduce the cost due to permanent failures, but it reduces the cost due to transient failures. Blake and Rodrigues [6] quantified the cost of a storage system due to permanent failures; extra replicas and/or erasure codes do not prevent this cost. Use of extra replicas is independent of erasure codes. Extra replicas and a minimum data availability threshold work with both systems using replication as well as systems using erasure codes.

Data placement is widely studied in the theory and systems community. For instance, the so-called "File allocation problem" [16] (FAP) discussed by Dowdy and Foster has frequently been formulated as an optimization problem. Dowdy and Foster survey over fifty papers in literature that formulate FAP as a proper optimization problem with formal constraints and an objective function, then apply standard optimization procedures to solve it. e.g. linear or integer programming, gradient descent, etc. More recently, Keeton has formulated storage configuration for data availability as an optimization problem for local area clusters. Similarly, FARSITE [15], has formulated data placement in the corporate environment as an optimization problem. Some systems attempt to group related nodes into clusters and place data replicas into separate clusters [32]. Finally, some algorithms attempt to place data on nodes that are as dissimilar as possible to avoid catastrophes [19]. The problem with the optimization, clustering, and catastrophe avoidance setups is that they are often NP-Hard, are performed offline, and usually are the result of a static analysis that considers past relationships and does not consider the complex dynamic interactions between nodes such as those caused by transient failures.

10 Conclusion

Storage systems have to replace lost redundancy due to permanent storage node failures. The significant problem in wide-area storage systems is that such systems experience high rates of transient failures due to Internet path outages, network partitions, software bugs, hardware failures, and so on; the problem is that it is not possible to differentiate permanent from transient failures. In this paper, we present a principled way of choosing data redundancy type, number of replicas, extra redundancy, and data placement. We found that existing storage systems are often configured to have low data availability or high maintenance bandwidth with analysis and trace-driven simulations of maintaining data on a long-term basis. Significant maintenance efficiency gains can be realized in existing storage systems with the correct choice of strategies and parameters.

As future work, we would like to apply our empirical study results to running systems. In particular, we want to modify deployed DHTs such as OpenDHT [20] to perform lazy repair and tune dynamically the data redundancy parameters based on changing system conditions.

References

[1] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Proc. of USENIX File and Storage Technologies (FAST)*, January 2002.

[2] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, , and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proc. of NSDI*, March 2004.

[3] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. In *Proc. of ACM SIGCOMM Conf.*, August 2002.

[4] R. Bhagwan, S. Savage, and G. Voelker. Replication strategies for highly available peer-to-peer storage systems. Technical Report CS2002-0726, U. C. San Diego, November 2002.

[5] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Totalrecall: Systems support for automated availability management. In *Proc. of NSDI*, March 2004.

[6] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proc. of HOTOS*, May 2003.

[7] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. of Sigmetrics*, June 2000.

[8] J. Cates. Robust and efficient data management for a distributed hash table. Master's thesis, MIT, June 2003.

[9] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. Prototype implementation of archival intermemory. In *Proc. of IEEE ICDE*, pages 485–495, February 1996.

[10] B. Chun and A. Vahdat. Workload and failure characterization on a large-scale federated testbed. Technical Report IRB-TR-03-040, Intel Research, November 2003.

[11] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proc. of ACM SIGCOMM Conf.*, August 2004.

[12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, October 2001.

[13] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *Proc. of NSDI*, March 2004.

[14] R. Dingledine, M. Freedman, and D. Molnar. The freehaven project: Distributed anonymous storage service. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[15] J. R. Douceur and R. P. Wattenhofer. Large-Scale Simulation of Replica Placement Algorithms for a Serverless Distributed File System. In *Proc. of MASCOTS*, 2001.

[16] L. W. Dowdy and Derrell V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, 1982.

[17] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP*, 2001.

[18] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *Proc. of ACM SOSP*, pages 29–43, October 2003.

[19] F. junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. M. Voelker. Surviving internet catastrophe. In *Proc. of USENIX Annual Technical Conf.*, May 2005.

[20] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring adoption of dhts with openhash, a public dht service. In *Proc. of IPTPS*, February 2004.

[21] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.

[22] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *Proc. of IPTPS*, February 2003.

[23] S. Nath, H. Yu, P.G. Gibbons, and S. Seshan. Tolerating correlated failures in wide-area monitoring services. Technical Report IRP-TR-04-09, Intel Research, May 2004.

[24] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach.* Forthcoming Edition.

[25] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. of USENIX File and Storage Technologies (FAST)*, 2003.

[26] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance free global storage in oceanstore. In *Proc. of IEEE Internet Computing*. IEEE, September 2001.

[27] R. Rodrigues and B. Liskov. High availability in dhts: Erasure

coding vs. replication. In *Proc. of IPTPS*, March 2005.

[28] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware*, November 2001.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM Conf.* ACM, August 2001.

[30] Jeremy Stribling. Planetlab all-pairs ping. http://infospect.planet-lab.org/pings.

[31] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, March 2002.

[32] H. Weatherspoon, T. Moscovitz, and J. Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proc. of Intl. Workshop on Reliable Peer-to-Peer Distributed Systems*, October 2002.

[33] H. Weatherspoon, C. Wells, and J. Kubiatowicz. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Proc. of Intl. Workshop on Future Directions of Distributed Systems*, 2002.

[34] P. Yalagandula, S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems. In *Proc. of USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, December 2004.

[35] H. Yu and A. Vahdat. Consistent and automatic replica regeneration. In *Proc. of NSDI*, March 2004.