Subtleties in Tolerating Correlated Failures

Abstract

High availability is widely accepted as an explicit requirement for distributed storage systems. Tolerating correlated failures is a key issue in achieving high availability in today's wide-area environments. This paper systematically revisits previously proposed techniques for addressing correlated failures. Using a combination of experimental and mathematical analysis of several real-world failure traces, we debunk four common myths about how to design systems to tolerate such failures. Based on our analysis, we identify a set of design principles that system builders can use to build services that tolerate correlated failures. We show how these lessons can be effectively used by incorporating them into ALI, a distributed read-write storage layer that provides high availability. Our results using ALI on PlanetLab over the past 8 months demonstrate its ability to withstand large correlated failures and meet preconfigured availability targets.

1 Introduction

High availability is widely accepted as an explicit requirement for distributed storage systems (e.g., [7, 8, 9, 13, 14, 16, 19, 35]). This is partly because these systems are often used to store important data, and partly because, as a result of systems advances, performance is no longer the only/primary limiting factor in the utility of distributed storage systems. Under the assumption of failure independence, replicating or erasure coding the data provides an effective way to mask individual node failures. In fact, virtually every distributed storage system today uses some form of replication or erasure coding.

In reality, the assumption of failure independence is rarely true. Node failures are typically correlated, with multiple nodes in the system failing (nearly) simultaneously. The size of these correlated failures can be quite large. For example, Akamai experienced large distributed denial-of-service (DDoS) attacks on its servers in May and June 2004 that resulted in many of its client sites being unavailable [1]. PlanetLab experienced four failure events during the first half of 2004 in which more than 35 nodes failed within a few minutes. Such large correlated failure events may have numerous causes, including system software bugs, DDoS attacks, virus/worm infections, node overload, and human errors. The impact of failure correlation on system unavailability is dramatic (i.e., by orders of magnitude) [5, 32]. As a result, tolerating correlated failures is a key issue in designing highly-available distributed storage systems.

Even though researchers have long been aware of correlated failures, most systems [7, 8, 9, 13, 14, 34, 35] are still evaluated and compared under the assumption of independent failures. This is primarily because of the difficulties in modeling and analyzing correlated failures. Because of such difficulties, the best design is selected under an evaluation based on independent failures and then some overprovisioning is added in the hopes of offsetting the negative effects of failure correlation.

Some researchers have also tried to take correlated failures directly into account [16, 18, 31], proposing two approaches. In the first approach, the system either monitors failures to predict the correlation pattern (i.e., which sets of nodes tend to fail together) [31], or simply assumes that nodes with the same software versions tend to fail together [18]. The system then places the replicas accordingly. The second approach makes simplifying assumptions about the sizes of correlated failures to make the availability analysis more tractable, and then configures the system accordingly. For example, Glacier [16] is configured based on only the maximum failure size.

Our goal in this paper is to design and implement a distributed read/write storage layer that provides high availability despite correlated node failures in non-P2P wide-area environments (such as PlanetLab and Akamai). Distributed storage is a fundamental building block for many decentralized systems including distributed file systems and distributed database systems. To this end, we study three real-world failure traces, and use a combination of experimental and mathematical analysis to evaluate the impact of realistic failure patterns on system designs.

We show that previously proposed approaches, although plausible, are less effective than one might hope under real-world failure correlation, often resulting in system designs that are far from optimal. Our study also reveals the subtleties that cause this discrepancy between the perception and the reality. These new findings lead to four design principles for tolerating correlated failures. Because our findings surprised us, we will refer to the previous perceptions as myths. While some of the findings are perhaps more surprising than others, note that *none* of the findings and design principles were explicitly identified or carefully quantified prior to our work. These design principles are applied and implemented in our highly-available read/write storage

layer called ALI.[1] Specifically, our study reveals the following four myths about tolerating correlated failures, along with the corresponding realities, overlooked subtleties, and design principles.

- **Myth 1: Correlated failures can be avoided using previously proposed failure pattern prediction techniques.** We find that avoiding correlated failures by predicting the failure pattern (as in Oceanstore [31]) provides *negligible* benefits in alleviating the negative effects of correlated failures in our real-world failure traces. The subtle reason is that the top 1% of correlated failures (in terms of size) have a dominant effect on system availability, and their failure patterns seem to be the most difficult to predict. Thus, the impact of correlated failures cannot be readily avoided through careful replica placement.

- **Myth 2: Simple modeling of failure sizes is adequate.** We find that considering only a single (maximum) failure size (as in Glacier [16]) leads to suboptimal system designs. Under the same level of failure correlation, the system configuration as obtained in [16] can be both overly-pessimistic for lower availability targets (thereby wasting resources) and overly-optimistic for higher availability targets (thereby missing the targets). While it is obvious that simplifying assumptions (such as assuming a single failure size) will always introduce inaccuracy, our contribution is to show that the inaccuracy can be dramatic (instead of negligible) in practical settings. For example, in our traces, assuming a single failure size leads to designs that either waste 2.4 times the needed resources or miss the availability target by 2 nines. Hence, more careful modeling is crucial. We propose using a *bi-exponential* model to capture the *distribution* of failure sizes, and show how this helps avoid overly-optimistic or overly-pessimistic designs.

- **Myth 3: Additional fragments/replicas are always effective in improving availability.** For popular $(n/2)$-out-of-n encoding schemes (used in OceanStore [19, 30] and CFS [9]), as well as majority voting schemes [28] over n replicas, it is well known that increasing n yields an *exponential* decrease in unavailability under independent failures. In contrast, we find that under real-world failure traces with correlated failures, additional fragments/replicas result in strongly *diminishing* returns in availability improvement for many schemes including the previous two. It is important to note that such diminishing return does *not* directly result from the simple presence of correlated failures. For example, we observe no diminishing return under Glacier's single-failure-size correlation model. Rather, in our real-world failure traces, the diminishing return arises due to the combined impact of correlated failures of different sizes. We further observe that the diminishing return effects are so strong that even *doubling or tripling n* provides only limited benefits after a certain point. These findings imply that the approach discussed earlier of designing for independent failures and then using over-provisioning (by increasing n) can easily fail to achieve availability targets in practice.

- **Myth 4: Superior designs under independent failures remain superior under correlated failures.** While it is well known that failure correlation always hurts system availability, it is also tempting to (implicitly) assume that failure correlation will decrease the availability of all system designs *roughly equally*, in either absolute or relative terms. Instead, we find that the effects of failure correlation on different designs are dramatically different. Thus selecting between two designs $\mathcal{D}$ and $\mathcal{D}'$ based on their availability under independent failures often leads to the wrong choice: $\mathcal{D}$ can be far superior under independent failures but far inferior under real-world correlated failures. For example, our results show that while 8-out-of-16 encoding achieves *1.5 more nines* of availability than 1-out-of-4 encoding under independent failures, it achieves *2 fewer nines* than 1-out-of-4 encoding under correlated failures. Thus, system designs must be explicitly evaluated under correlated failures.

The findings from this paper depend unavoidably on the failure traces we used. Among the four findings above, the first one may be the most dependent on the specific traces. The other three findings, on the other hand, are likely to hold as long as failure correlation is non-trivial and has a wide range of failure sizes.

We have incorporated these lessons and design principles into our ALI prototype: ALI does not try to avoid correlated failures by predicting the correlation pattern; it uses a failure size distribution model rather than a single failure size; and it explicitly quantifies and compares configurations via online simulation with our correlation model. As an example application, we built a publicly-available distributed wide-area network monitoring system on top of ALI, and deployed it on over 450 PlanetLab nodes. ALI has been running over PlanetLab for the last 8 months. We summarize our performance and availability experience with ALI, reporting its behavior during a highly unstable period of PlanetLab (right before the SOSP'05 deadline), and demonstrating its ability to reach a pre-configured availability target.

[1] Named for boxer Mohammed Ali's ability to get hit with powerful punches yet remain standing.

2 Background and Related Work

Distributed storage systems and erasure coding. Distributed storage systems [7, 8, 9, 13, 14, 16, 19, 35] have long been an active area in systems research. Of these systems, only OceanStore [19] and Glacier [16] explicitly consider correlated failures. OceanStore uses a distributed hash table (DHT) based design to support a significantly larger user population (e.g., 10^{10} users) than previous systems. Glacier is a more robust version of the PAST [25] DHT-based storage system that is explicitly designed to tolerate correlated failures.

Distributed storage systems commonly use data redundancy (replication, erasure coding [24]) to provide high availability. In erasure coding, a data object is encoded into *n fragments*, out of which any *m* fragments can reconstruct the object. OceanStore uses $(n/2)$-out-of-*n* erasure coding. This configuration of erasure coding is also used by the most recent version of CFS [9, 12], one of the first DHT-based, read-only storage systems. Glacier, on the other hand, *adapts* the settings of *m* and *n* to achieve availability and resource targets. Replication can be viewed as a special case of erasure coding where $m = 1$.

In large-scale systems, it is often desirable to automatically create new fragments upon the loss of existing ones (due to node failures). We call such systems *regeneration* systems. Almost all recent distributed storage systems (including OceanStore, Glacier, CFS, and our ALI system) are regeneration systems. Under the assumption of failure independence, the availability of regeneration systems is typically analyzed [35] using a Markov chain to depict the birth-death process.

Data maintenance costs. In regeneration systems, different data redundancy designs not only result in differing availability, but also differing data maintenance costs (i.e., differing amounts of data transferred over the network in regenerating the objects). There has been much previous work on data maintenance costs. In particular, Weatherspoon et al. [29] comprehensively study such costs in recent decentralized wide-area storage systems. Their study shows that the maintenance costs of random replica placement (with small optimizations) are similar to those of an idealized optimal placement where failure correlation is completely avoided.

In comparison, our paper focuses on the effects of failure correlation on availability instead of maintenance costs. Many optimizations for decreasing maintenance costs (such as lazy repair, high/low watermarks, and techniques to minimize data transfers during transient/temporary node failures) are orthogonal to the designs and results in this paper. Finally, the results from [29] do not conflict with our conclusion that random replica placement (or even more sophisticated placement) cannot effectively alleviate the negative effects of failure correlation on *availability*.

Previous availability studies of correlated failures. Traditionally, researchers assume failure independence when studying availability [7, 8, 9, 13, 14, 34, 35]. For storage systems, correlated failures have recently drawn more attention in the context of wide-area environments [10, 16, 18, 31, 32], local-area and campus network environments [8, 27], and disk failures [5, 11]. Below, we focus on those works most relevant to our study that were not fully discussed in Section 1.

The Phoenix Recovery Service [18] proposes placing replicas on nodes running heterogeneous versions of software, to better guard against correlated failures. Because their target environment is a heterogeneous environment such as a peer-to-peer system, their approach does not conflict with our findings.

The effects of correlated failures on erasure coding systems have also been studied [5] in the context of survivable storage disks. The study is based on availability traces of desktops [8] and different Web servers. (We use the same Web server trace in our study.) However, because the target context in [5] is disk drives, the study considers much smaller-scale systems (at most 10 disks) than we do.

None of the previous studies point out the myths and design principles in this paper or quantify their effects.

3 Methodology

This study is based on mathematical analysis, system implementation, and experimental evaluation. The experiments use a combination of three testbeds: live PlanetLab deployment, real-time emulation on Emulab [2], and event-driven simulation. Each testbed allows progressively more extensive evaluation than the previous one and the results from one testbed help validate the results from the next testbed. We will report availability and performance results from the 8 month live deployment of our system over PlanetLab. We will also present detailed simulation results for deeper understanding into system availability, especially when exploring design choices. Simulation allows us to directly compare different system configurations by subjecting them to identical streams of failure/recovery events, which is not possible using our live deployment. We have carefully validated the accuracy of our simulation results by comparing them against the results from PlanetLab deployment and Emulab emulation. Our detailed comparison [23] shows that there is negligible difference in the three sets of results.

In this paper, we use ERASURE(m,n) to denote an *m*-out-of-*n* read-only erasure coding system. Also, to unify terminology, we often refer to replicas as fragments of an ERASURE$(1,n)$ system. Unless otherwise mentioned, all designs we discuss in this paper use regeneration to compensate for lost fragments due to node failures.

Trace	Duration	Nature of nodes	# of nodes	Probe interval	Probe method
PL_trace [3]	03/2003 to 06/2004	PlanetLab nodes	277 on avg	15 to 20 mins	all-pair pings; 10 ping packets per probe
WS_trace [5]	09/2001 to 12/2001	Public web servers	130	10 mins	HTTP GET from a CMU machine
RON_trace [4]	03/2003 to 10/2004	RON testbed	30 on avg	1 to 2 mins	all-pair pings; 1 ping packet per probe

Table 1: Three traces used in our study.

For the purpose of studying correlated failures, a *failure event* (or simply *failure*) crashes one or more nodes in the system. The number of nodes that crash is called the *size* of the failure. To distinguish a failure event from the failures of individual nodes, we explicitly call the latter *node failures*. A data object is *unavailable* if it can not be reconstructed due to node failures. We present availability results using standard "number of nines" terminology (i.e., $\log_{10}(1/\phi)$, where ϕ is the probability the data object is unavailable).

Failure traces. We use three real-world wide-area failure traces (Table 1) in our study. WS_trace is intended to be representative of public-access machines that are maintained by different administrative domains, while PL_trace and RON_trace potentially describe the behavior of a centrally administered distributed system that is used mainly for research purposes, as well as for a few long running services.

A probe *interval* is a complete round of all pair-pings or Web server probes. The PL_trace[2] and RON_trace traces consist of periodic probes between every pair of nodes. Each probe consists of multiple pings; we declare that a node has *failed* if *none* of the other nodes can ping it during that interval. We do not distinguish between whether the node has failed or has simply been partitioned from all other nodes—in either case it is unavailable to the overall system. The WS_trace trace contains logs of HTTP GET requests from a single source node at CMU to multiple Web servers. Our evaluation of this trace is not as precise because near-source network partitions make it appear as if all the other nodes have failed. To mitigate this effect, we assume our probing node is disconnected from the network if 4 or more consecutive HTTP requests to different servers fail.[3] We then ignore all failures during that probe period. Note that this heuristic may still not perfectly classify source and server failures, but we believe that the error is likely to be minimal.

In studying correlated failures, each probe interval is considered as a separate failure event whose size is the number of failed nodes that were available during the previous interval. More details on our trace processing methodology can be found in [23].

[2]Note that PL_trace has sporadic "gaps", see [23] for how we classify the gaps and treat them properly based on their causes.

[3]This threshold is the smallest to provide a plausible number of near-source partitions. Using a smaller threshold would imply that the client (on Internet2) experiences a near-source partition $> 4\%$ of the time in our trace, which is rather unlikely.

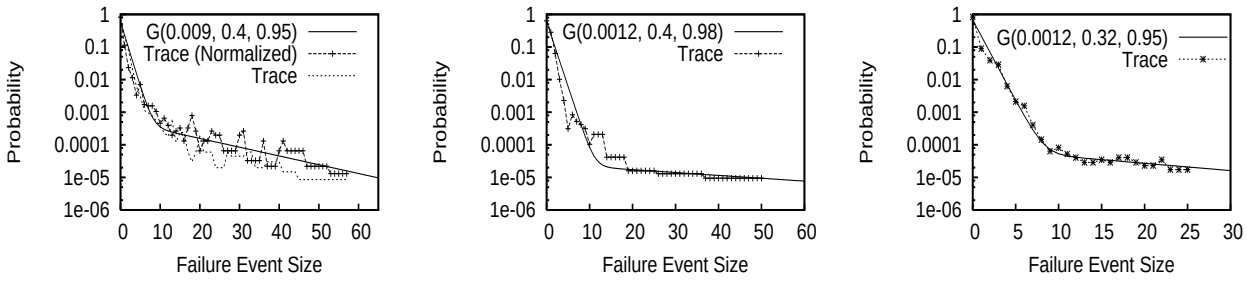
Limitations. Although the findings from this paper depend on the traces we use, the effects observed in this study are likely to hold as long as failure correlation is non-trivial and has a wide range of failure sizes. One possible exception is our observation about the difficulty in predicting failure patterns of larger failures. However, we believe that the sources of such large failures (e.g., DDoS attacks) make accurate prediction difficult in any deployment.

Given that the target of ALI is non-P2P environments, we intentionally study failure traces from systems with largely homogeneous software (operating systems, etc.). Even in WS_trace, at the time of the trace over 78% of the web servers were using the same Apache server running over Unix, according to the historical data at http://www.netcraft.com. We believe that wide-area non-P2P systems (such as Akamai) will largely be homogeneous because of the prohibitive overhead of maintaining multiple versions of software. Failure correlation in P2P systems can be dramatically different from other wide-area systems, because many failures in P2P systems are due to user departures. It is part of our future work to extend our study to P2P environments. Another limitation of our traces is that the long probe interval prevents the detection of short-lived failures. This makes our availability results slightly optimistic.

Steps in our study. Section 4 constructs a tunable failure correlation model from our three failure traces; this model allows us to study the sensitivity of our findings beyond the three traces. Sections 5–7 present the four myths, and their corresponding realities, overlooked subtleties, and design principles. These sections focus on read-only ERASURE(m,n) systems, and mainly use trace-driven simulation, supplemented by model-driven simulation for sensitivity study. Section 8 shows that our conclusions for read-only systems readily extend to read/write systems. Section 9 describes and evaluates ALI, including its availability running on PlanetLab for the past 8 months.

4 A Tunable Model for Correlated Failures

This section constructs a tunable parameterized failure correlation model from the three failure traces. The primary purpose of this model is to allow sensitivity studies and experiments with correlation levels that are stronger or weaker



(a) PL_trace (b) WS_trace (c) RON_trace

Figure 1: Correlated failures in three real-world traces. $G(\alpha, \rho_1, \rho_2)$ is our correlation model.

than in the traces. In addition, the model also later enables us to avoid overly-pessimistic and overly-optimistic designs, as well as to perform analytical studies for deeper understanding. The model balances idealized assumptions (e.g., Poisson arrival of correlated failure events) with realistic characteristics (e.g., mean-time-to-failure and failure size distribution) extracted from the real-world traces. In particular, it aims to accurately capture large (but rare) correlated failures, which have a dominant effect on system unavailability.

4.1 Correlated Failures in Real Traces

We start by investigating the failure correlation in our three traces. Figure 1 plots the PDF of failure event sizes for the three traces. Because of the finite length of the traces, we cannot observe events with probability less than 10^{-5} or 10^{-6}. While RON_trace and WS_trace have a roughly constant node count over their entire duration, there is a large variation in the total number of nodes in PL_trace. To compensate, we use both raw and normalized failure event sizes for PL_trace. The normalized size is the (raw) size multiplied by a normalization factor γ, where γ is the number of nodes in the interval divided by the average number of nodes (i.e., 277) in PL_trace.

In all traces, Figure 1 shows that failure correlation has different strengths in two regions. In PL_trace, for example, the transition between the two regions occurs around event size 10. In both regions, the probability decreases roughly exponentially with the event size. However, the probability decreases significantly faster for small-scale correlated failures than for large-scale ones. We call such a distribution *bi-exponential*. Although we have only anecdotal evidence, we conjecture that different failure causes are responsible for the different parts of the distribution. For example, we believe that system instability, some application bugs, and localized network partitions are responsible for the small failure events. It is imaginable that the probability decreases quickly as the scale of the failure increases. On the other hand, human interference, attacks, viruses/worms and large ISP failures are likely to be responsible for large failures. This is supported by the fact that many of the larger PlanetLab failures can be attributed to DDoS attacks (e.g., on 12/17/03), system software bugs (e.g., on 3/17/04), and node overloads (e.g., on 5/14/04). Once such a problem reaches a certain scale, extending its scope is not much harder. Thus the probability decreases relatively slowly as the scale of the failure increases.

4.2 A Tunable Bi-Exponential Model

In our basic failure model, failure events arrive at the system according to a Poisson distribution. The entire system has a *universe* of u nodes. The model does not explicitly specify which of the u nodes each failure event crashes, for the following reasons: Predicting failure patterns is not effective in improving availability, and pattern-aware fragment placement achieves almost identical availability as a pattern-oblivious random placement (see Section 5). Thus, our availability study needs to consider only the case where the m fragments of a data object are placed on a random set of m nodes. Such a random placement, in turn, is equivalent to using a fixed set of m nodes and having each failure event (with size s) crash a random set of s nodes in the universe. Realizing the above point helps us to avoid the unnecessary complexity of modeling which nodes each failure event crashes – each failure event simply crashes a random set of s nodes.

We find that many existing distributions (such as heavy-tail distributions [21]) cannot capture the bi-exponential property in the trace. Instead, we use a model that has two exponential components, one for each region. Each component has a tunable parameter ρ between 0 and ∞ that intuitively captures the slope of the curve and controls how strong the correlations are. When $\rho = 0$, failures are independent, while $\rho = \infty$ means that every failure event causes the failure of all u nodes. Specifically, for $0 < \rho < \infty$, we define the following geometric sequence: $f(\rho, i) = c(\rho) \cdot \rho^i$. The normalizing factor $c(\rho)$ serves to make $\sum_{i=0}^{u} f(\rho, i) = 1$.

We can now easily capture the bi-exponential property by

composing two $f(\rho,i)$. Let p_i be the probability of failure events of size i, for $0 \leq i \leq u$. Our correlation model, denoted as $G(\alpha,\rho_1,\rho_2)$, defines $p_i = (1-\alpha)f(\rho_1,i) + \alpha f(\rho_2,i)$, where α is a tunable parameter that describes the probability of large-scale correlated failures. Compared to a piece-wise function with different ρ's for the two regions, $G(\alpha,\rho_1,\rho_2)$ avoids a sharp turning point at the boundary between the two regions.

Figure 1 shows how well this model fits the three traces. The parameters of the models are chosen such that the Root Mean Square errors between the model and the trace points are minimized. Because of space limitations, we are only able to provide a brief comparison among the traces here. The parameters of the model are different across the traces, in large part because the traces have different universe sizes (10 failures out of 277 is quite different from 10 failures out of 30). As an example of a more fair comparison, we selected 130 random nodes from `PL_trace` to enable a comparison with `WS_trace`, which has 130 nodes. The resulting trace is well-modeled by $G(0.009,0.3,0.96)$. This means that the probability of large-scale correlated failures in `PL_trace` is about 8 times larger than `WS_trace`.

Failure arrival rate and recovery. Up to this point, the correlation model $G(\alpha,\rho_1,\rho_2)$ only describes the failure event size distribution, but does not specify the event arrival rate. To study the effects of different levels of correlation, the event arrival rate should be such that the average mean-time-to-failure (MTTF) of nodes in the traces is always preserved. Otherwise with a constant failure event arrival rate, increasing the correlation level would have the strong side effect of decreasing node MTTFs. To preserve the MTTF, we determine the system-wide failure event arrival rate λ to be such that $1/(\lambda\sum_{i=1}^{u}(ip_i)) = \text{MTTF}/u$.

We observe from the traces that, in fact, there exists non-trivial correlation among node recoveries as well. Moreover, nodes in the traces have non-uniform MTTR and MTTF. However, our experiments show that the above two factors have little impact on system availability for our study. Specifically, for all parameters we tested (e.g., later in Figure 4), the availability obtained under model-driven simulation (which assumes independent recoveries and uniform MTTF/MTTR) is almost identical to that obtained under trace-driven simulation (which has recovery correlation and non-uniform MTTF/MTTR). Therefore, our model avoids the unnecessary complexity of modeling recovery correlation and non-uniform MTTF/MTTR.

Stability of the model. We have also performed an extensive stability study for our model using `PL_trace` and `RON_trace`. We do not use `WS_trace` due to its short length. We summarize our results below – see [23] for the full results. Our results show that each model converges within a 4 month period in its respective failure traces. We believe this convergence time is quite good given the rarity of large correlation events. Second, the model built ("trained") using a prefix of a trace (e.g., the year 2003 portion) reflects well the failures occurring in the rest of the trace (e.g., the year 2004 portion). This is important because we later use the model to configure ALI to provide a given availability target.

5 Myth: Correlated Failures Can Be Avoided Using Previous Failure Pattern Prediction Techniques

We present our four myths starting from this section. Unless otherwise stated, all our results here are obtained via detailed event-driven simulation (including data regeneration) based on the three real failure traces. The bi-exponential model is used only when we need to tune the correlation level—in such cases we will explicitly mention its use. Because many existing systems (e.g., OceanStore, Glacier, and CFS) use a large number of fragments, we show results for up to $n = 60$ fragments (as well as for large values of m).

The Myth. Chun et al. [10] point out that node failure histories can be used to discover a relatively stable pattern of correlated failures (i.e., which set of nodes tend to fail together), based on a fraction of the `PL_trace` used in this study. Weatherspoon et al. [31] (as part of the OceanStore project [19]) reach a similar conclusion by analyzing a four-week failure trace of 306 web servers[4]. Based on such predictability, they further propose a framework for online monitoring and clustering of nodes. Nodes within the same cluster are highly correlated, while nodes in different clusters are more independent. They show that the clusters constructed from the first two weeks of their trace are similar to those constructed from the last two weeks. Given the stability of the clusters, they conjecture that by placing the n fragments (or replicas) in n different clusters, the n fragments will not observe excessive failure correlation among themselves. In some sense, the problem of correlated failures goes away.

The Reality. We revisit this technique by using the same method as in [31] to process our failure traces. For each trace, we use the first half of the trace (i.e., "training data") to cluster the nodes using the same clustering algorithm [26] as used in [31]. Then as a case study, we consider two placement schemes of an ERASURE$(n/2,n)$ (as in OceanStore) system. The first scheme (*pattern-aware*) explicitly places the n fragments of the same object in n different clusters, while the second scheme (*pattern-oblivious*) simply places

[4] The trace actually contains 1909 web servers, but the authors analyzed only 306 servers because those are the only ones that ever failed during the four weeks.

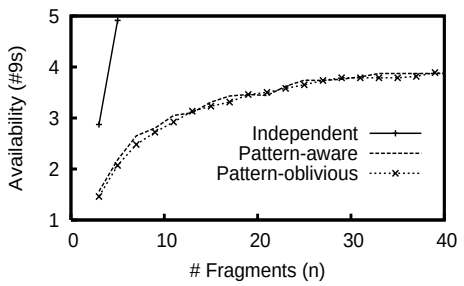


Figure 2: Negligible availability improvements from failure pattern prediction of WS_trace for ERASURE($n/2,n$) systems.

the fragments on n random nodes. Finally, we measure the availability under the second half of the trace.

We first observe that most ($\approx$ 99%) of the failure events in the second half of the traces affect only a very small number (≤ 3) of the clusters computed from the first half of the trace. This implies that the clustering of correlated nodes is relatively stable over the two halves of the traces, which is consistent with [31].

On the other hand, Figure 2 plots the achieved availability of the two placement schemes under WS_trace. PL_trace shows similar results [23]. (We do not use RON_trace because it contains too few nodes for the clustering to be meaningful.) The graph shows that explicitly choosing different clusters to place the fragments gives us *negligible* improvement on availability. We also plot the availability achieved if the failures in WS_trace were independent. This is done via model-driven simulation and by setting the parameters in our bi-exponential model accordingly. For a fair comparison, we ensure that the machine MTTF and MTTR (and hence the machine unavailability) in the model match the MTTF and MTTR in WS_trace. Note that when the failures are independent, the two placement schemes do not make any difference. The large differences between the curve for independent failures and the other curves show that there are strong negative effects from failure correlation in the trace. Identifying and exploiting failure patterns, however, has almost no effect in alleviating such impacts. We have also obtained similar findings under a wide-range of other m and n values for ERASURE(m,n).

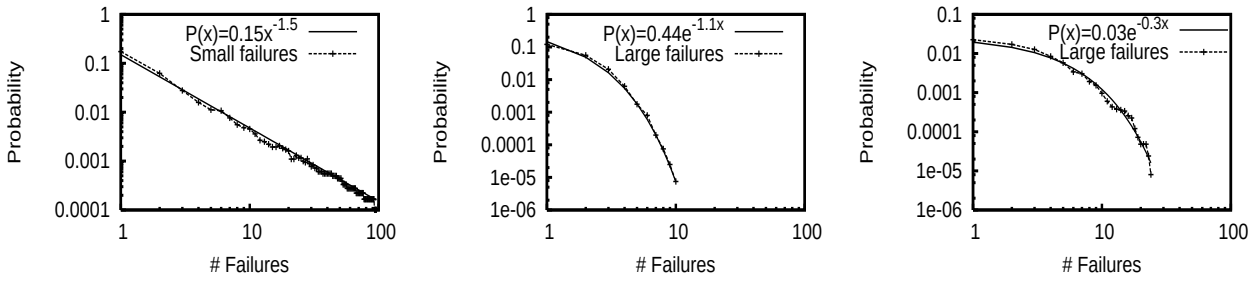
A natural question that arises is whether the above findings are because our traces are different from the traces studied in [10, 31]. Our PL_trace is, in fact, a superset of the failure trace used in [10]. On the other hand, the failure trace studied in [31], which we call Private_trace, is not publicly available. Is it possible that Private_trace gives even better failure pattern predictability than our traces, so that pattern-aware placement would indeed be effective? To answer this question, we directly compare the "pattern predictability" of the traces using the metric from [31]: the *average mutual information among the clusters* (MI) (see [31] for a rigorous definition). A smaller MI means that the clusters constructed from the training data predict the failure patterns in the rest of the trace better. Weatherspoon et al. report an MI of 0.7928 for their Private_trace. On the other hand, the MI for our WS_trace is 0.7612. This means that the failure patterns in WS_trace are actually more "predictable" than in Private_trace.

The Subtlety. To understand the above seemingly contradictory observations, we take a deeper look at the failure patterns. To illustrate, we classify the failures into small failures and large failures based on whether the failure size exceeds 15. With this classification, in all traces, most ($\approx$ 99%) of the failures are small.

Next, we investigate how accurately we can predict the failure patterns for the two classes of failures. We use the approach used in [17] for showing that UNIX processes running for a long time are more likely to continue to run for a long time in future. For a random pair of nodes in WS_trace, Figure 3(a) plots the probability that the pair crashes together (in correlation) more than x times because of the small failures. The straight line in log-log scale indicates that the data fits the Pareto distribution of $P(\#failure \geq x) = cx^{-k}$, in this case for $k = 1.5$. We observe similar fits for PL_trace ($P(\#failure \geq x) = 0.3x^{-1.45}$) and RON_trace ($P(\#failure \geq x) = 0.02x^{-1.4}$). Such a fit to the Pareto distribution implies that pairs of nodes that have failed together many times in the past are more likely to fail together in the future [17]. Therefore, past pairwise failure patterns (and hence the clustering) caused by the small failures are likely to hold in the future.

Next, we move on to large failure events (Figure 3(b)). Here, the data fit an exponential distribution of $P(\#failure \geq x) = ae^{-bx}$. The memoryless property of the exponential distribution means that the frequency of future correlated failures of two nodes is independent of how often they failed together in the past. This in turn means that we cannot easily (at least using existing approaches) predict the failure patterns caused by these large failure events. Intuitively, large failures are generally caused by external events (e.g., DDoS attacks), occurrences of which are not predictable in trivial ways. We observe similar results in the other two traces. For example, Figure 3(c) shows an exponential distribution fit for the large failure events in RON_trace. For PL_trace, the fit is $P(\#failure \geq x) = 0.3e^{-0.9x}$.

Thus, the pattern for roughly 99% of the failure events (i.e., the small failure events) is predictable, while the pattern for the remaining 1% (i.e., the large failure events) is not easily predictable. On the other hand, our experiments show that large failure events, even though they are only 1% of all failure events, contribute the most to un-



(a) Small failures in `WS_trace` (b) Large failures in `WS_trace` (c) Large failures in `RON_trace`

Figure 3: Predictability of pairwise failures. For two nodes selected at random, the plots show the probability that the pair will fail together (in correlation) more than *x* times during the entire trace. Distributions fitting the curves are also shown.

availability. For example, the unavailability of a "pattern-aware" ERASURE$(16,32)$ under `WS_trace` remains unchanged (0.0003) even if we remove all the small failure events. The intuition behind this result is that because small failure events only affect a small number of nodes, those failure events can be almost completely masked by data redundancy and regeneration. This explains why on one hand, failure patterns are largely predictable, while on the other hand, pattern-aware fragment placement is not effective in improving availability. It is also worth pointing out that the sizes of these large failures still span a wide range (e.g., from 15 to over 50 in `PL_trace` and `WS_trace`). So capturing the distribution over all failure sizes is still important in the bi-exponential model.

The Design Principle. *Large correlated failures (which comprise a small fraction of all failures) have a dominant effect on system availability, and system designs must not overlook these failures. For example, because failure pattern prediction that works well for the bulk of correlated failures fails for large correlated failures, it is not effective in alleviating the negative effects of correlated failures.*

Discussion. In certain scenarios the patterns of large failures can be predicted. For example, if we have several LAN clusters each with thousands of nodes, then clearly the patterns of some large failures (e.g., power outages that crash a whole cluster) can be predicted. In such cases, it is beneficial to place the fragments on different LANs. The same is true in P2P systems where failures are actually user leaves, which follow a clear diurnal pattern. On the other hand, in wide-area non-P2P systems, at least in our traces, there does not seem to exist large-scale strong correlations that would make the pattern of large failures predictable in this manner.

6 Myth: Simple Modeling of Failure Sizes Is Adequate

The Myth. A key challenge in system design is to obtain the "right" set of parameters that will be neither overly-pessimistic nor overly-optimistic in achieving given design goals. Because of the complexity in availability estimation introduced by failure correlation, system designers sometimes make simplifying assumptions on correlated failure sizes in order to make the problem more amenable. For example, Glacier [16] considers only the (single) maximum failure size, aiming to achieve a given availability target despite the correlated failure of up to a fraction f of all the nodes. This simplifying assumption enables the system to use a closed-form formula [16] to estimate availability and then calculate the needed m and n values in ERASURE(m,n).

The Reality. Figure 4 plots the number of fragments needed to achieve given availability targets under Glacier's model (with $f = 0.65$ and $f = 0.45$) and under `WS_trace`. Glacier does not explicitly explain how f can be chosen in various systems. But at least we can expect that f should be a constant under the same deployment context (e.g., for `WS_trace`).

A critical point to observe in Figure 4 is that for the real trace, the curve is not a straight line (we will explore the shape of this curve later). Because the curves from Glacier's estimation are roughly straight lines, they always significantly depart from the curve under the real trace, regardless of how we tune f. For example, when $f = 0.45$, Glacier over-estimates system availability when n is large: Glacier would use ERASURE$(6,32)$ for an availability target of 7 nines, while in fact, ERASURE$(6,32)$ only achieves slightly above 5 nines availability. Under the same f, Glacier also under-estimates the availability of ERASURE$(6,10)$ by roughly 2 nines. If f is chosen so conservatively (e.g., $f = 0.65$) that Glacier never over-estimates, then the under-estimation becomes even more significant. As a result, Glacier would suggest ERASURE$(6,31)$ to achieve 3

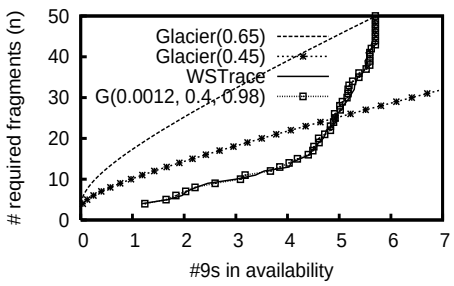


Figure 4: Number of fragments in ERASURE$(6,n)$ needed to achieve certain availability targets, as estimated by Glacier's single failure size model and our distribution-based model of G(0.0012,0.4,0.98). We also plot the actual achieved availability under the trace.

nines availability while in reality, we only need to use ERASURE$(6,9)$. This would unnecessarily increase both the storage required for an object and the bandwidth used to create or update the object by over 240%.

While it is obvious that simplifying assumptions (such as assuming a single failure size) will always introduce inaccuracy, our above results show that the inaccuracy can be dramatic (instead of negligible) in practical settings.

The Subtlety. The reason behind the above mismatch between Glacier's estimation and the actual availability under `WS_trace` is that in real systems, failure sizes may cover a large range. In the limit, failure events of any size may occur; the only difference is their likelihood. System availability is determined by the combined effects of failures with different sizes. Such effects cannot be summarized as the effects of a series of failures of the same size (even with scaling factors).

To avoid the overly-pessimistic or overly-optimistic configurations resulting from Glacier's method, a system must consider a distribution of failure sizes. ALI uses the bi-exponential model for this purpose. Figure 4 also shows the number of fragments needed for a given availability target as estimated by our simulator driven by the bi-exponential model. The estimation based on our model matches the curve from `WS_trace` quite well. It is also important to note that the difference between Glacier's estimation and our estimation is purely from the difference between single failure size and a distribution of failure sizes. It is not because Glacier uses a formula while we use simulation. In fact, we have also performed simulations using only a single failure size, and the results are similar to those obtained using Glacier's formula.

The Design Principle. *Assuming a single failure size can result in dramatic rather than negligible inaccuracies in practice. Thus correlated failures should be modeled via a distribution instead of a maximum failure size.*

7 Impact of Failure Correlation

Systems researchers have long been aware of the negative effects of correlated failures. However, systematically evaluating a design under correlated failures is quite difficult. Given such difficulty, system designs are often selected based on their evaluation under independent failures and then some over-provisioning is added in the hopes of offsetting the negative effects of failure correlation. This section discusses two myths associated with this approach.

7.1 Myth: Additional Fragments Are Always Effective in Improving Availability

The Myth. Under independent failures, distributing the data always helps to improve availability, and any target availability can be achieved by distributing fragments across more and more machines, without increasing the storage overhead. For this reason, system designers sometimes fix the ratio between n and m (so that the storage overhead, n/m, is fixed), and then simply increase n to achieve the required availability target. As an example, in OceanStore, n/m is always kept at a constant of 2, and OceanStore targets better availability by increasing n. For instance, under independent failures, ERASURE$(16,32)$ gives much better availability than ERASURE$(12,24)$, which, in turn, gives better availability than ERASURE$(8,16)$. CFS uses a similar ERASURE$(n/2,n)$ scheme in its design. Given independent failures, it can even be proved [23] that increasing the number of fragments (while keeping storage constant) *exponentially* decreases unavailability. Thus, we can plausibly over-provision an ERASURE(m,n) system using a larger n to offset the negative effects of failure correlation, without increasing the storage overhead.

There is a similar myth regarding read/write replication systems (i.e., over-provisioning by adding more replicas). Here the storage overhead increases with over-provision. Later in Section 8, we will show that a read/write replication system using majority voting [28] for consistency has the same availability as ERASURE$(n/2,n)$. Thus our discussion in this section also applies to read/write replication systems.

The Reality. We will show that, perhaps surprisingly, over-provisioning is not effective under correlated failures, even if we *double* or *triple* n. Figures 5 and 6 plot the availability of ERASURE$(4,n)$, ERASURE$(n/3,n)$ and ERASURE$(n/2,n)$ under `WS_trace` and `PL_trace`, respectively, as a function of n. We do not use `RON_trace` because it contains only 30 nodes. Both ERASURE$(n/3,n)$ and ERASURE$(n/2,n)$ suffer from a strong diminishing return effect. For example, increasing n from 20 to 60 in ERASURE$(n/2,n)$ provides less than a half nine's improvement. On the other hand, ERASURE$(4,n)$ does not suffer

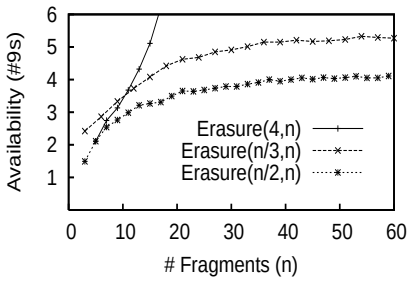


Figure 5: Availability of ERASURE(m,n) under `WS_trace`

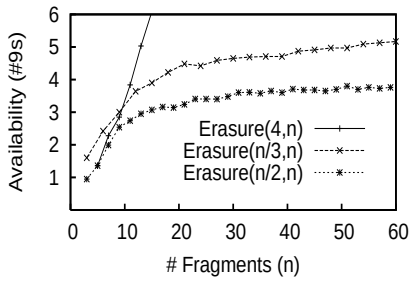


Figure 6: Availability of ERASURE(m,n) under `PL_trace`

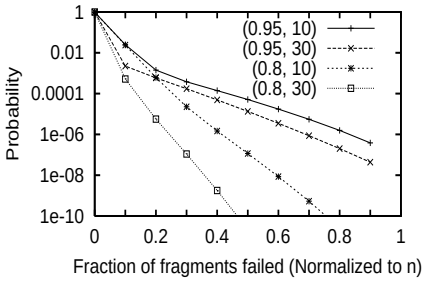


Figure 7: Fraction of fragments failed for $u = 277$, $\alpha = 0.009$ and $\rho_1 = 0.4$. The legends are in the form of (ρ_2, n).

from such an effect. By tuning the parameters in the correlation model and using model-driven simulation, we further observe [23] that the diminishing return effects become more prominent under stronger correlation levels as well as under larger m values.

It is important to note that such diminishing return does *not* directly result from the simple presence of correlated failures. For example, we observe no diminishing return under Glacier's single-failure-size correlation model (Figure 4 and also see [23] for additional results). Rather, in our real-world failure traces, the diminishing return arises due to the combined impact of correlated failures of different sizes.

The above diminishing return shows that correlated failures prevent a system from effectively improving availability by over-provisioning (without increasing storage). In read/write systems using majority voting, the problem is even worse: the same diminishing return effect occurs even allowing for significant increases in storage.

The Subtlety. To find out the subtle cause behind the previous results, we analyze system availability under correlated failures using our bi-exponential model $G(\alpha,\rho_1,\rho_2)$. The detailed mathematical derivation of the results is available separately in [23]. Let $P(j,n)$ be the probability that a failure event on a universe of u nodes causes exactly j failures among the n fragments ($0 \le j \le n \le u$). We have:

$$P(j,n) = (1-\alpha)\cdot h(\rho_1,j,n) + \alpha \cdot h(\rho_2,j,n), \qquad (1)$$

where

$$h(\rho,j,n) = \frac{\binom{n}{j}(1-\rho)\rho^j}{1-\rho^{u+1}} \sum_{k=0}^{u-n} \rho^k \frac{\binom{u-n}{k}}{\binom{u}{k+j}}$$

A closer examination of this formula reveals a subtle, yet fundamental effect of correlated failures. Based on equation 1, Figure 7 plots the probability that a single failure event causes a certain fraction of node failures within the n nodes holding the n fragments, for two different values of ρ_2. ERASURE$(n/2,n)$ is able to mask all the failure events that cause less than 50% of the fragments to fail, meaning that only the probabilities of the failure events causing more than 50% fragment failures matter. When $\rho_2 = 0.8$, increasing n from 10 to 30 significantly decreases the probability that over 50% of the fragments will all fail. However, when $\rho_2 = 0.95$, as in `PL_trace`, tripling n only slightly decreases this probability. The intuition for this is as follows. Although more fragments allow the system to tolerate more failures, they also mean that the scope of the vulnerability has increased – the same failure event may cause more failures within the n fragments. Such effects of a larger *vulnerability scope* become stronger when the correlation level increases. Ultimately, when $\rho_2 = 0.95$, such negative effects almost completely offset the benefits of being able to tolerate a larger absolute number of fragment failures.

To directly relate $P(j,n)$ to availability, we next provide a first-order approximation of the unavailability of ERASURE$(n/2,n)$ by (1) considering that a single failure event that fails at least half the fragments is needed to initiate unavailability, and (2) ignoring that additional failure events may arrive while waiting to completely recover from this initial failure event:

$$U(n/2,n) \approx \lambda \sum_{j=n/2}^{n} P(j,n) \sum_{i=n/2}^{j} \frac{\text{MTTR}}{i}, \qquad (2)$$

where $U(n/2,n)$ is the unavailability of ERASURE$(n/2,n)$ and λ is defined in Section 4.2.

From the above two equations, it can be shown [23] that for an integer x that is not too large, if $P(xj,xn) \approx P(j,n)$, then $U(xn/2,xn) \approx U(n/2,n)$. This means that under certain correlation levels (so that $P(xj,xn) \approx P(j,n)$ holds, as in our earlier example), even doubling or tripling the number of fragments will not be effective in improving availability under ERASURE$(n/2,n)$.

The only way to lessen these diminishing return effects is to non-trivially increase the n/m ratio (i.e., the storage overhead) as we increase n, by keeping m small. When m is small, the previous analysis no longer holds. Thus, unlike ERASURE$(n/2,n)$, ERASURE$(4,n)$, for example, is able to mask an increasingly large fraction of failed fragments as n

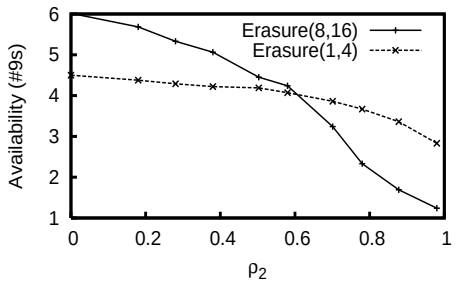


Figure 8: Effects of the correlation model $G(0.0012, \frac{2}{5}\rho_2, \rho_2)$ on two systems. $\rho_2 = 0$ indicates independent failures, while $\rho_2 = 0.98$ indicates roughly the correlation level observed in `WS_trace`.

increases, as confirmed by Figures 5 and 6.

The Design Principle. *System designers should be aware that correlated failures result in strong diminishing return effects in* ERASURE*(m,n) systems unless m is kept small. For popular systems such as* ERASURE*(n/2,n), even doubling or tripling n provides very limited benefits after a certain point.*

7.2 Myth: Superior Designs under Independent Failures Remain Superior under Correlated Failures

The Myth. While it is well known that failure correlation always hurts system availability, it is also tempting to (implicitly) assume that failure correlation will decrease the availability of all system designs *roughly equally*, in either absolute (e.g., by an absolute amount of 0.01) or relative (e.g., by a relative amount of 10%) terms. If this were indeed true, then we could plausibly still choose designs base on an analysis under independent failures. Namely, a design that was superior to other designs under independent failures would roughly remain superior to those designs under correlated failures.

The Reality. We find that, unfortunately, this is not always true—correlated failures hurt some designs much more than others. Such non-equal effects are demonstrated via the example in Figure 8. Here, we plot the unavailability of ERASURE$(1,4)$ and ERASURE$(8,16)$ under different failure correlation levels, by tuning the parameters in our correlation model in model-driven simulation. These experiments use the model $G(0.0012, \frac{2}{5}\rho_2, \rho_2)$ (a generalization of the model for `WS_trace` to cover a range of ρ_2), and a universe of 130 nodes, with MTTF = 10 days and MTTR = 1 day. ERASURE$(1,4)$ achieves around 1.5 fewer nines than ERASURE$(8,16)$ when failures are independent (i.e., $\rho_2 = 0$). On the other hand, under the correlation level found in `WS_trace` ($\rho_2 = 0.98$), ERASURE$(1,4)$ achieves 2 more nines of availability than ERASURE$(8,16)$.[5]

The Subtlety. The cause of this (perhaps counter-intuitive) result is the diminishing return effect described earlier. As the correlation level increases, ERASURE$(8,16)$, with its larger *m*, suffers from the diminishing return effect to a much greater degree than ERASURE$(1,4)$. In general, because diminishing return effects are stronger for systems with larger *m*, correlated failures hurt systems with large *m* more than those with small *m*.

The Design Principle. *A superior design under independent failures may not be superior under correlated failures. In particular, correlation hurts systems with larger m more than those with smaller m. Therefore, system designs should be explicitly evaluated and compared under correlated failures.*

8 Read/Write Systems

Thus far we have discussed the four myths in the context of read-only ERASURE(m,n) systems. Interestingly, our results extend quite naturally to read/write systems that use quorum techniques or voting techniques to maintain data consistency.

Quorum systems (or voting systems) [6, 28] are standard techniques for maintaining consistency for read/write data. We first focus on the case of pure replication. Here a user accessing a data object needs to coordinate with a *quorum* of replicas. For example, the majority quorum system (or majority voting) [28] requires that the user coordinates with a majority of the replicas. This ensures that any reader intersects in at least one replica with the latest writer, so that the reader sees the latest update. If a majority of the replicas is not available, then the data object is unavailable to the user. From an availability perspective, this is exactly the same as a read-only ERASURE$(n/2,n)$ system. Among all quorum systems that always guarantee consistency, we will consider only majority voting because its availability is provably optimal [6].

Recently, Yu [34] proposed *signed quorum systems* (SQS) where quorum sizes are smaller than $n/2$, at the cost of a tunably small probability of reading stale data. Smaller quorum sizes help to improve system availability because fewer replicas need to be available for a data object to be available. If we do not consider regeneration (i.e, creating new replicas to compensate for lost replicas), then the availability of such SQS-based systems would be exactly the same as that of a read-only ERASURE(m,n) system, where *m* is the quorum size. Regeneration makes the problem slightly more complex, but our design principles still apply [23].

[5] The same conlusion also holds if we directly use `WS_trace` to drive the simulation.

Finally, quorum systems can also be used over erasure-coded data [15]. Despite the complexity of these protocols [15], they all have simple threshold-based requirements on the number of available fragments. As a result, their availability can also be readily captured by properly adjusting m in ERASURE(m,n) in our results.

9 ALI: A Highly-Available Read/Write Storage Layer

Applying the design principles in the previous sections, we have built ALI, a decentralized read/write storage layer that is highly-available despite correlated failures. ALI does not try to avoid correlated failures by predicting correlation pattern. Rather, it tolerates them by choosing the right set of parameters. ALI determines system parameters using a model with a failure size distribution rather than a single failure size. Finally, ALI explicitly quantifies and compares configurations via online simulation using our correlation model. ALI allows both replication and erasure coding for data redundancy, and also implements both majority quorum systems and SQS for data consistency.

We have incorporated ALI as the read-write storage layer of NETMON (name anonymized), a wide-area network monitoring system deployed on over 450+ PlanetLab nodes. NETMON has been available for public use for the last 8 months.

At a high level, ALI consists of two major components. The first component replicates or encodes objects, and processes normal case user accesses. The second component is responsible for regeneration, and automatically creates new fragments when existing ones fail. These two subsystems have similar designs as Om [35]. For lack of space, we here omit the details and focus on two aspects that are more unique to ALI.

9.1 Achieving Target Availability

Applications configure ALI by specifying an *availability target*, a *bi-exponential failure correlation model*, as well as a *cost function*. The correlation model can be specified by saying that the deployment context is "PlanetLab-like," "WebServer-like," or "RON-like." In these cases, ALI will use one of the three built-in failure correlation models from Section 4. We also intend to add more built-in failure correlation models in the future. To provide more flexibility, ALI also allows the application to directly specify the three tunable parameters in the bi-exponential distribution. It is our long term goal to extend ALI to monitor failures in the deployment, and automatically adjust the correlation model if the initial specification is not accurate.

The cost function is an application-defined function that specifies the overall cost resulting from performance overhead and inconsistency (for read/write data). It takes three inputs, m, n, and i (for inconsistency), and returns a cost value that the system intends to minimize given that the availability target is satisfied. For example, a cost function may bound the storage overhead (i.e., n/m) by returning a high cost if n/m exceeds certain threshold. Similarly, the application can use the cost function to ensure that not too many nodes need to be contacted to retrieve the data (i.e., bounding m). We choose to leave the cost function to be completely application-specific because the requirements from different applications can be dramatically different.

With the cost function and the correlation model, ALI uses online simulation to determine the best values for m, n, and quorum sizes. It does so by exhaustively searching the parameter space (with some practical caps on n and m), and picking the configuration that minimizes the cost function while still achieving the availability target. The amount of inconsistency (i) is predicted [33, 34] based on the quorum size. Finally, this best configuration is used to instantiate the system. Our simulator takes around 7 seconds for each configuration (i.e., each pair of m and n values) on a single 2.6GHz Pentium 4; thus ALI can perform a brute-force exhaustive search for 20,000 configurations (i.e., a cap of 200 for both n and m, and $m \leq n$) in about one and a half days. Many optimizations are possible to further prune the search space. For example, if ERASURE$(16,32)$ does not reach the target, then ERASURE$(17,32)$ can never reach the target either. This exhaustive search is performed offline; its overhead does not affect system performance.

9.2 Optimizations for Regeneration

When regenerating read/write data, ALI (like RAMBO [22] and Om [35]) uses the Paxos distributed consensus protocol [20] to ensure consistency. Using such consistent regeneration techniques in the wide-area, however, poses two practical challenges that are not addressed by RAMBO and Om (RAMBO has never been deployed over the wide-area, while Om has never been evaluated/tested under a large number of simultaneous node failures):

Flooding problem. After a large correlated failure, one instance of the Paxos protocol needs to be executed for each of the objects that have lost any fragments. For example, our NETMON deployment on PlanetLab has 3530 objects storing PlanetLab sensor data, and each object has 7 replicas. A failure of 42 out of 206 PlanetLab nodes (on 3/28/2004) would flood the system with 2,478 instances of Paxos. Due to the message complexity of Paxos, this creates excessive overhead and stalls the entire system.

Positive feedback problem. Determining whether a distant node has failed is often error-prone due to unpredictable

communication delays and losses. Regeneration activity after a correlated failure increases the inaccuracy of failure detection due to the increased load placed on the network and nodes. Unfortunately, inaccurate failure detections trigger more regeneration which, in turn, results in larger inaccuracy. Our experience shows that this positive feedback loop can easily crash the entire system.

To address the above two problems, ALI implements two optimizations:

Opportunistic Paxos-merging. In ALI, if the system needs to invoke Paxos for multiple objects that *happen* to reside on the same set of nodes, the regeneration module merges all the related Paxos invocations into a single one. This approach is particularly effective with ALI's load balancing algorithm (beyond the scope of the paper), which tends to place two objects either on exactly the same set of nodes or on completely disjoint sets of nodes. Compared to explicitly aggregating objects into clusters, this approach avoids the need to maintain consistent split and merge operations on clusters.

Paxos admission-control. To avoid the positive feedback problem, we use a simple admission control mechanism on each node to control its CPU and network overhead, and to avoid excessive false failure detections. Specifically, each node, before initiating a Paxos, samples (by piggybacking on the periodic ping messages) the number of ongoing Paxos instances on the relevant nodes. Paxos is initiated only if the average and the maximum number of ongoing Paxos instances are below some thresholds (2 and 5, respectively, in our current PlanetLab deployment). Otherwise, the node queues the Paxos instance and backs off for a small random time before trying again. Immediately before a queued Paxos is started, ALI rechecks whether regeneration is still necessary (i.e., whether there still are failures). This further improves failure detection accuracy, and also avoids regeneration when the failed nodes have already recovered.

9.3 Availability of ALI in the Wild

In this section and the next, we report our availability and performance experience with ALI (as part of NETMON) on PlanetLab over the past 8 months. NETMON uses ALI to maintain 3530 objects for different sensor data collected from PlanetLab nodes. As a background testing and logging mechanism, we have continuously issued one query to NETMON every five minutes since it was deployed. Each query tries to access all the objects stored in ALI. We set a target availability of 99.9% for our deployment, and ALI chooses, accordingly, a replica count of 7 and a quorum size of 2.

Figure 9 plots the behavior of our deployment under stress over a 62-hour period before the deadline of a major systems conference. PlanetLab tends to suffer the most failures and unstabilities right before major conference deadlines. The figure shows both the fraction of available nodes in PlanetLab and the fraction of available (i.e., with accessible quorums) objects in ALI.

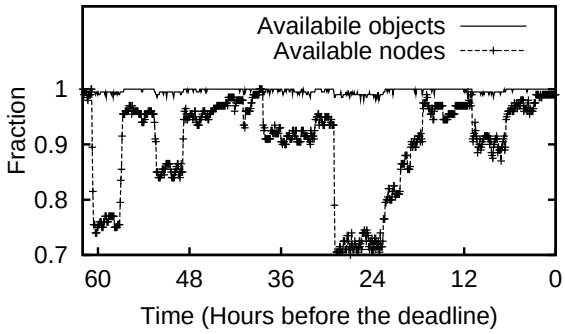


Figure 9: Availability of NETMON in the wild.

At the beginning of the period, ALI was running on around 200 Planetlab nodes. Then PlanetLab experienced a few large correlated failures, which correspond to the sharp drops in the "available nodes" curve (the lower curve). On the other hand, the fluctuation of the "available objects" curve (the upper curve) is consistently small, and is always above 98% even during such an unusual stress period. This means that our real world ALI deployment tolerated these large correlated failures well. In fact, the overall availability achieved by ALI during the 8 month deployment was roughly 99.9%, demonstrating the ability of ALI to achieve a pre-configured target availability.

9.4 Basic Performance of ALI

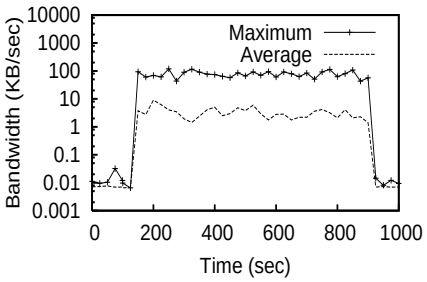
Basic latency for accessing data. Figure 10(a) shows the measured latency from our lab to access a single object in our NETMON deployment. The object is replicated in random PlanetLab nodes. The table represents two scenarios based on whether the replicas are chosen only from the USA (first row) or from all over the world (second row). The significant part of the latencies and the high standard deviations come mainly from network latency.

Bandwidth usage. Our next experiment studies the amount of bandwidth consumed by regeneration, including the Paxos protocol. To do this, we consider the PlanetLab failure event on 3/28/2004 that caused 42 out of the 206 live nodes to crash in a short period of time (an event found by analyzing `PL_trace`). We replay this event by deploying NETMON on 206 PlanetLab nodes and then killing the NETMON process on 42 nodes.

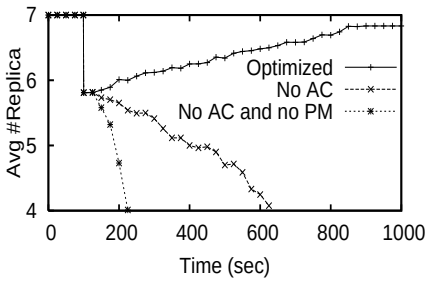
Replica Location	Average Latency (ms)			Std. Dev.
	Network	Processing	Total	(ms)
USA	60	22	82	68
WORLD	165	25	190	238

(Averages and Std. Deviations are computed from 50 independent data accesses)

(a) End-to-end latency



(b) Bandwidth consumption



(c) Avg. # of replicas per object

Figure 10: Basic performance of ALI. In (c), "AC" means Paxos admission-control, "PM" means Paxos-merging.

Figure 10(b) plots the bandwidth used during regeneration. The failures are injected at time 100. We observe that on average, each node only consumes about 3KB/sec bandwidth, out of which 2.8KB/sec is used to perform necessary regeneration, 0.27KB/sec for unnecessary regeneration (caused by false failure detection), and 0.0083KB/sec for failure detection (pinging). The worst-case peak for an individual node is roughly 100KB/sec, which is sustainable even with home DSL links.

Regeneration time. Finally, we study the amount of time needed for regeneration, demonstrating the importance of our Paxos-merging and Paxos admission-control optimizations. We consider the same failure event (i.e., killing 42 out of 206 nodes) as above.

Figure 10(c) plots the average number of replicas per object and shows how ALI gradually regenerates failed replicas after the failure at time = 100s. In particular, the failure affects 2478 objects. Without Paxos admission-control or Paxos-merging, the regeneration process does not converge.

Paxos-merging reduces the total number of Paxos invocations from 2478 to 233, while admission-control helps the regeneration process to converge. Note that the average number of replicas does not reach 7 even at the end of the experiment because some objects lose a majority of replicas and cannot regenerate. A single regeneration takes 21.6sec on average which is dominated by the time for data transfer (14.3sec) and Paxos (7.3sec). The time for data transfer is determined by the amount of data and can be much larger. The convergence time of around 12 minutes is largely determined by the admission-control parameters and may be tuned to be faster. However, regeneration is typically not a time-sensitive task, because ALI only needs to finish regeneration before the next failure hits. Our simulation study based on `PL_trace` shows that 12-minute regeneration time achieves almost identical availability as, say, 5-minute regeneration time. Thus we believe ALI's regeneration mechanism is adequately efficient.

10 Conclusion

Despite the wide awareness of correlated failures in the research community, the properties of such failures and their impact on system behavior is poorly understood. In this paper, we have made some critical steps toward helping system developers both to understand the impact of realistic, correlated failure patterns on their systems and to design systems that tolerate such failures. Our study demonstrated that common perceptions on how to tolerate correlated failures under real-world conditions are often flawed. Our discussion of the subtleties of correlated failures revealed lessons for system builders and provided a set of design principles to follow. We presented the design and implementation of a distributed read/write storage layer, ALI, that uses these design principles to meet availability targets even in the event of real-world correlated failures.

References

[1] Akamai Knocked Out, Major Websites Offline. http://techdirt.com/articles/20040524/0923243.shtml.

[2] Emulab - network emulation testbed home. http://www.emulab.net.

[3] PlanetLab - All Pair Pings. http://www.pdos.lcs.mit.edu/∼strib/pl_app/.

[4] The MIT RON trace. David G. Andersen, Private communication.

[5] BAKKALOGLU, M., WYLIE, J., WANG, C., AND GANGER, G. On Correlated Failures in Survivable Storage Systems. Tech. rep., CMU, May 2002. SCS Technical Report CMU-CS-02-129.

[6] BARBARA, D., AND GARCIA-MOLINA, H. The Reliability of Voting Mechanisms. *IEEE Trans. Comput.* (October 1987), 1197–1208.

[7] BHAGWAN, R., TATI, K., CHENG, Y., SAVAGE, S., AND VOELKER, G. M. TotalRecall: Systems support for automated availability management. In *NSDI* (2004).

[8] BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *SIGMETRICS* (2000).

[9] CATES, J. Robust and Efficient Data Management for a Distributed Hash Table. Masters Thesis, Massachusetts Institute of Technology, May 2003.

[10] CHUN, B., AND VAHDAT, A. Workload and failure characterization on a large-scale federated testbed. Tech. Rep. IRB-TR-03-040, Intel Research Berkeley, November 2003.

[11] CORBETT, P., ENGLISH, B., GOEL, A., GRCANAC, T., KLEIMAN, S., LEONG, J., AND SANKAR, S. Row-diagonal parity for double disk failure correction. In *USENIX FAST* (2004).

[12] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area Cooperative Storage with CFS. In *ACM SOSP* (2001).

[13] DABEK, F., LI, J., SIT, E., ROBERTSON, J., KAASHOEK, M. F., AND MORRIS, R. Designing a DHT for low latency and high throughput. In *NSDI* (2004).

[14] DOUCEUR, J. R., AND WATTENHOFER, R. P. Competitive Hill-Climbing Strategies for Replica Placement in a Distributed File System. In *DISC* (2001).

[15] GOODSON, G., WYLIE, J., GANGER, G., AND REITER, M. Efficient Byzantine-tolerant Erasure-coded Storage. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)* (June–July 2004).

[16] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *NSDI* (2005).

[17] HARCHOL-BALTER, M., AND DOWNEY, A. B. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems 15*, 3 (1997), 253–285.

[18] JUNQUEIRA, F., BHAGWAN, R., HEVIA, A., MARZULLO, K., AND VOELKER, G. Surviving internet catastrophes. In *USENIX Annual Technical Conference* (2005).

[19] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *ASPLOS* (2000).

[20] LAMPORT, L. The Part-Time Parliament. *ACM Transactions on Computer Systems 16* (May 1998), 133–169.

[21] LELAND, W. E., TAQQ, M. S., WILLINGER, W., AND WILSON, D. V. On the self-similar nature of Ethernet traffic. In *ACM SIGCOMM* (1993).

[22] LYNCH, N., AND SHVARTSMAN, A. RAMBO: a reconfigurable atomic memory service for dynamic networks. In *DISC* (2002).

[23] *Names removed for anonymous submission.* Subtleties in Tolerating Correlated Failures. Tech. rep., 2005. An anonymized version can be obtained from the Program Chair.

[24] PLANK, J. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software – Practice & Experience 27*, 9 (September 1997), 995–1012.

[25] ROWSTRON, A., AND DRUSCHEL, P. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *ACM SOSP* (2001).

[26] SHI, J., AND MALIK, J. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell. 22*, 8 (2000), 888–905.

[27] TANG, D., AND IYER, R. K. Analysis and Modeling of Correlated Failures in Multicomputer Systems. *IEEE Transactions on Computers 41*, 5 (May 1992), 567–577.

[28] THOMAS, R. H. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems 4* (1979), 180–209.

[29] WEATHERSPOON, H., CHUN, B.-G., SO, C. W., AND KUBIATOWICZ, J. Long-term data maintenance: A quantitative approach. Tech. Rep. UCB/CSD-05-1404, University of California, Berkeley, 2005.

[30] WEATHERSPOON, H., AND KUBIATOWICZ, J. Erasure coding vs. replication: A quantitative comparison. In *IPTPS* (2002).

[31] WEATHERSPOON, H., MOSCOVITZ, T., AND KUBIATOWICZ, J. Introspective Failure Analysis: Avoiding Correlated Failures in Peer-to-Peer Systems. In *Proceedings of International Workshop on Reliable Peer-to-Peer Distributed Systems* (October 2002).

[32] YALAGANDULA, P., NATH, S., YU, H., GIBBONS, P. B., AND SESHAN, S. Beyond Availability: Towards a Deeper Understanding of Machine Failure Characteristics in Large Distributed Systems. In *WORLDS* (2004).

[33] YU, H. Overcoming the Majority Barrier in Large-Scale Systems. In *DISC* (2003).

[34] YU, H. Signed Quorum Systems. In *PODC* (2004).

[35] YU, H., AND VAHDAT, A. Consistent and Automatic Replica Regeneration. *ACM Transactions on Storage 1*, 1 (February 2005).