Introspective Failure Analysis: Avoiding Correlated Failures in Peer-to-Peer Systems

Hakim Weatherspoon, Tal Moscovitz and John Kubiatowicz
Computer Science Division
University of California, Berkeley
{hweather, mtal, kubitron}@cs.berkeley.edu
http://oceanstore.cs.berkeley.edu

Abstract

Failure independence is an important assumption for many fault tolerance techniques. Unfortunately, real system exhibit correlated failures. In this paper, we present a framework for online discovery of groups of server nodes that are maximally independent in their failure characteristics. We discuss the framework in detail and provide a preliminary evaluation.

1 Introduction

Many systems are based on simplifying assumptions. Making such assumptions aids in reasoning about these system, but can fail to capture important aspects of actual system behavior. One important example is that of reliability metrics. Many fault tolerant algorithms are designed under the assumption that no more than an explicit fraction of components can fail [11]. This characterization implicitly assumes that the probability of a component failing while a protocol is in progress is independent of the duration of the protocol, that all components have an identical probability of failure, and that components fail independently. These assumptions do not adequately reflect the nature of real-world network environments. In practice, failures are correlated. Correlated faults can lead to reduced system availability or reliability [10].

This paper makes the following contributions. First, we give two examples in which fault tolerance can be increased by using independently failing components. Second, we present failure analysis framework that allow nodes that fail with low correlation to be chosen as the set of n machines. Finally, we show preliminary results when this technique is used to cluster related web servers together.

2 M of N Fault Tolerance

Redundancy is the simplest technique used to increase reliability in P2P systems. Redundancy is the use of multiple resources (computations, copies of data, etc.) when a smaller subset or single one would suffice. We call the general use of redundant components m of n fault tolerance; that is, when *any* m (out of n) components function correctly, the system behaves correctly. In this section, we show two examples of this type of fault tolerance.

Byzantine Fault Tolerance Byzantine Agreement protocols [13] allow a set of servers to come to a unified decision, even if some of them (less than $\frac{1}{3}$) are actively attempting to compromise the process. As an example, OceanStore [12] and Farsite [2] use Byzantine Agreement during the serialization of updates. In the terminology of this section, we can say that Byzantine Agreement functions correctly when any $m = 2f + 1$ out of $n = 3f + 1$ nodes are uncompromised. If more than $n - m = f$ nodes are compromised, the system stops functioning correctly. By selecting nodes that are sufficiently uncorrelated in their behavior (operating characteristics, software, ownership, *etc*), we can minimize the chance that an adversary could compromise the decision process.

Erasure Coding *Erasure codes* provide redundancy without the overhead of strict replication [4, 16, 6]. Erasure codes encode an object into n *fragments*, any m of which are sufficient to reconstruct the object ($m < n$). We call $r = \frac{m}{n} < 1$ the *rate* of encoding. A rate r code increases the storage cost by a factor of $\frac{1}{r}$. For example, an $r = \frac{1}{4}$ encoding might produce $n = 64$ fragments, any $m = 16$ of which are sufficient to reconstruct data, with a total overhead factor of *four*. Note that $m = 1$ represents strict replication, and RAID level 5 can be described by ($m = 4$, $n = 5$). Figure 1 illustrates the durability of a block of data encoded

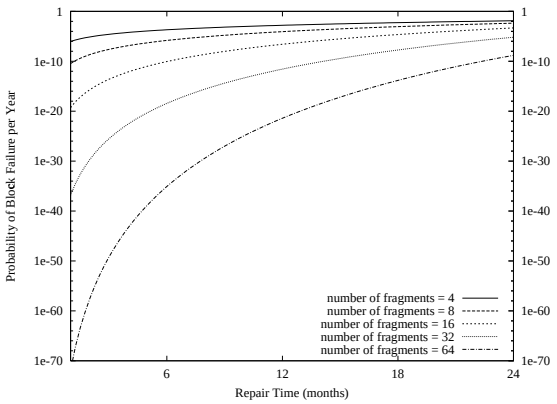

Figure 1. Fraction of Blocks Lost Per Year (FBLPY) for a rate $\frac{1}{4}$, erasure-encoded block. *Here, disks fail after five years and a repair process reconstructs data regularly. The four-fragment case (top line) is equivalent to simple replication on four servers. Increasing number of fragments drastically increases the durability of a block while total storage overhead remains constant. Notice, for example, that for a repair interval of 6 months the four-way replication (top line) loses 0.03 (3%) of blocks per year while the 64 fragments, any 16 of which are sufficient to reconstruct (bottom line) loses* 10^{-35} *of blocks per year.*

in an erasure code, measured in Fraction of Blocks Lost Per Year (FBLPY). Each line represents a different degree of fragmentation (n) at the same total overhead – illustrating that increased fragmentation provides greatly increased durability [19].

Failure Independence Both of the previous examples assume that failures are *independent* and *identically distributed*. This is not true in general. Server failures may be correlated because they share network routers, software bugs, configuration problems, operating systems, *etc.*. The rate of failure may be elevated in regions of the network that share common administration or unstable hardware elements. Studies have shown that human errors and network problems are major causes for node failures [14]. An increased rate of failure can be addressed through greater redundancy (*i.e.* more components), while increased correlation can be defeated through selective use of resources. We explore measurement and modeling techniques in the following sections that can be used to choose a set of nodes that are maximally independent.

3 Dissemination Architecture

In this section, we present a framework for discovering sets of nodes that fail with low correlation. We call

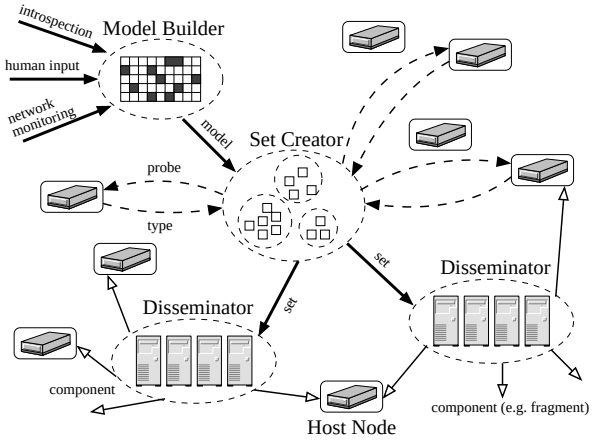

Figure 2. *Four components of dissemination framework.*

the subsequent placement of elements on such servers *dissemination*. This framework is part of the OceanStore system [12], but could be utilized with an array of current systems, including Farsite [2], Intermemory [4], CFS [5], and PAST [7].

The framework is composed of three parts. The *Model Builder* takes input from human sources, network measurements, and online observation to develop a model of failure correlation. The *Dissemination Set Creator* discovers nodes in the network and utilizes the results of the Model Builder to generate *dissemination sets*, sets of nodes that fail with low correlation. The *Disseminator* sends one component to each node in a set. These nodes then perform their desired function (*e.g.* store fragments or perform Byzantine Agreement). Figure 2 shows this framework. We describe each of these parts thoroughly in Sections 4, 5, and 6.

This framework has a couple of important performance characteristics. First, the Model Builder and the Set Creator can operate asynchronously from the Disseminator. Thus, the computation and network latency penalties incurred by the Model Builder and the Set Creator do not impact the dissemination latency. Second, dissemination sets constructed by the Creator can be used by many Disseminators. Thus, the network and computational resources are amortized over several Disseminators.

4 Model Builder

The Model Builder is responsible for developing models that predict the correlation of failure among *types* of nodes. A type is a tuple enumerating a server's properties with respect to a number of failure *dimensions*. A dimension is any property that affects the availability of components such as geographic location, administrative domain, operating system type and release, and IP subnet. The Model Builder

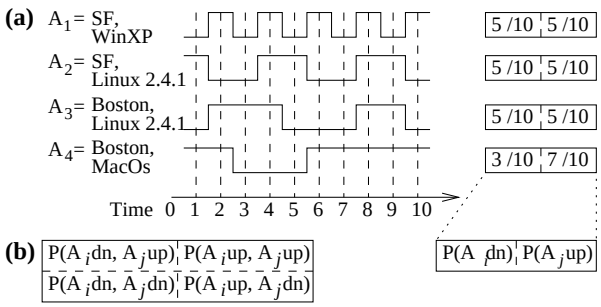

Figure 3. *(a) A sample history of availability for four different types of machines is observed. The marginal probabilities are also shown. (b) The pair-wise joint probabilities from the example history.*

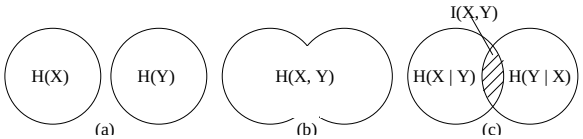

Figure 4. *(a)* H(X) *is the* entropy *of* X. *Entropy is a measure of uncertainty. The more random a variable is, the more entropy it will have. (b)* H(X,Y) *is the* joint entropy. *The joint entropy is the combined uncertainty of two random variables. (c)* H(X|Y) *is the* conditional entropy. *The conditional entropy of* X *given the value of* Y. Mutual Information, I(X,Y), *is a measure of the reduction of the entropy of* X *given knowledge of* Y

collects information about various dimensions from human sources, network monitoring [1], and observations.

Implementation Rather than searching the entire node space for a maximally independent set of nodes, the problem of finding nodes that fail independently would be more tractable if we grouped highly correlated nodes together and consider each group a *domain*. Grouping correlated nodes into domains permits simpler analysis and more compact models. Instead of simulating complex relationships between different nodes, we can profile *types* of nodes.

The Model Builder collects availability statistics on different machine types. Figure 3(a) shows an example with ten data points for each of four machine types. Profiling *types* of machines greatly reduces the number of machines in the system that need to be monitored (i.e. type of machines $\ll$ number of machines in system). From the simple data collection, we can compute marginal and pair-wise joint probabilities of failure (i.e. uptime, downtime, or combinations) as in Figures 3(a) and 3(b). After data has been collected, there are many ways to build a model. For instance the Chi Square Test for Independence can be used to test for degree of independence between any two types. A Bayesian view may also be appropiate – there are several possible interpretations in the context of prior world knowledge. The difficulty is in specifying the prior world knowledge. Also, many well known clustering algorithms can be used to compute *domains*, highly correlated, on type of machines. Examples of applying such schemes have appeared in literature; such as, *mutual information*[3].

There are many other techniques available for building these models; we are currently examining the viability of the alternatives. A common mechanism used to build these models is to setup a weighted graph $G = (V, E)$ and set the weight on the edge connecting two nodes to be a measure of the similarity between two nodes. This graph is also known as a *similarity matrix* We show one metric for edge weights below and how we use the edge weights to build domains in Section 5

Mutual Information as a Graph Edge Weight We use mutual information to perform comprehensive pairwise comparisons to produce an edge weight connecting two nodes. The medical field has used mutual information in the genomic project to find functional genomic clusters in RNA expression data [3]. The mutual information is a measure of the additional information known about one type of machine when given another, as shown in equation 1. Entropy H is the measure of uncertainty in a random variable. Mutual Information is the reduction of that uncertainty given another random variable.

$$I(X, Y) = H(X) - H(X|Y) \tag{1}$$

where $H(X)$ is the *entropy* of X, *entropy* is a measure of randomness, and $H(X|Y)$ is the *conditional entropy*. Mutual information is also defined in terms of joint and marginal probability distributions as in equation 2

$$I(X, Y) = \sum_{x \in X} \sum_{y \in Y} p(x, y) log \frac{p(x, y)}{p(x)p(y)} \tag{2}$$

where $X, Y \in up, down$. A mutual information at zero means that the joint distribution of expression values hold no more information than the types of machines considered separately. In this way, mutual information can be used as a metric between two types of machines related by their degree of independence. It is hypothesized that the higher the mutual information is between two types, the more likely it is they have the same failure relationship. We use mutual information to group highly correlated types of machines into domains and assert that type of machines in different domains have low correlation.

Other graph Edge Weights In addition to mutual information as a distance metric for correlation, we used the joint pair-wise probability and the joint pair-wise log probability. The pair-wise probability is the probability that the nodes fail at the same.

5 Set Creator

In this section we discuss the *Set Creator*. To create dissemination sets, the Set Creator must first collect information about a sufficiently large set of nodes. The Creator relies on the properties of a *Decentralized Object Location and Routing* (DOLR) system such as CAN [15], Chord [18], Pastry [7], or Tapestry [8], to discover new nodes. A DOLR routes messages to nodes that most closely matches the requested destination. Thus, even though the DOLR address space is sparse, the Set Creator can use a scan of the address space to reach a random set of nodes. Servers that are willing to host a specified function(e.g. store fragments, participate in a byzantine agreement, etc) respond with a securely *signed* statement of their type.

Once the Creator has collected a sufficiently large pool of types of nodes (e.g. several hundred), it analyzes the probability for correlated failures using the result of the Model Builder and creates sets of type of nodes that fail with low correlation. Resulting dissemination sets should be large enough that each component of a function may be sent to a different node in the set with a few additional nodes to replace failed nodes. Because the model is constantly evolving and the properties of nodes are constantly changing, dissemination sets eventually expire. If no sets of sufficient size can be found, the Set Creator seeks out other nodes to replace some members of the pool.

6 Disseminator

The *Disseminator* sends one component to each node in a *dissemination set* and waits for signed acknowledgements. If acks are received for all components, dissemination is complete. If acks for some components are not received, the process must decide if enough components were acknowledged to consider the dissemination effective. If not, the Disseminator may re-send unacknowledged components to the extra members of the set or redisseminate completely. Note that while the Disseminator normally uses recently-created dissemination sets, it may reuse old, unexpired sets if necessary. Computing the m of n coding scheme and network latency are the only performance bottleneck for the Disseminator since the dissemination set's are pre-computed by a *Set Creator* as described in Section 5.

7 Experimental Setup

Our experimental setup consisted of 1909 web servers located throughout the world. We extracted each nodes characteristics through a combination of the *whois* network program, netcraft[1], and telnet. The node properties that we collected were administrative domain, operating system, geographic location, and web server program. We collected uptime statistics through netcraft[1], as netcraft has a contractual[1] agreement with the site owner to monitor the site for uptime. The uptime statistics were collected on a daily scale; that is, we queried the sites uptime on a daily basis and if the sites uptime was less than 24 hours the site was considered not available during some interval in time the previous day. We conducted our study for two weeks.

After we collected the uptime data, we performed three qualitative evaluations. First, we evaluated whether the clusterings made sense; that is, we did a sanity check to see if highly related nodes were placed in the same cluster. For example, web servers from microsoft.com in the same building should be clustered together. Second, we compared the different clustering algorithms to each other using the same metric for comparison. Third, we seperated the two weeks of data into a training set (i.e. the model) and test set, first and second week, respectfully. Then we compared the test set to the model with our objective function to minimize intercluster similarity. That is, the average distance (edge weight) between a random node in one cluster and a random node in a different cluster is minimized.

Clustering We have implemented two different clustering algorithms: hierarchical clustering[9] and normalized cuts[17]. There are many other types of clustering algorithms to choose from, we chose these two to see how a greedy linear running time algorithm (hierarchical clustering) would compare to an algorithm that does what we want (i.e. simultaneously minimizing intercluster similarity and maximizing intracluster similarity), but is possibly NP-Complete. Normalized cuts approximates by setting up the analysis as an eigenvector problem. Hierarchical clustering can create solutions that have local minimas but are not optimal globally. Normalized cuts is used in the vision community to partition and segment images.

8 Preliminary Results

In this section we present preliminary evaluation for the dissemination architecture using both hierarchical and normalized cuts clustering, both with mutual information and joint probability as a metric (the edge weight). Also, we

[1]The contract occurs automatically when the site owner queries netcraft to be monitored. Netcraft monitors over 38 million sites.

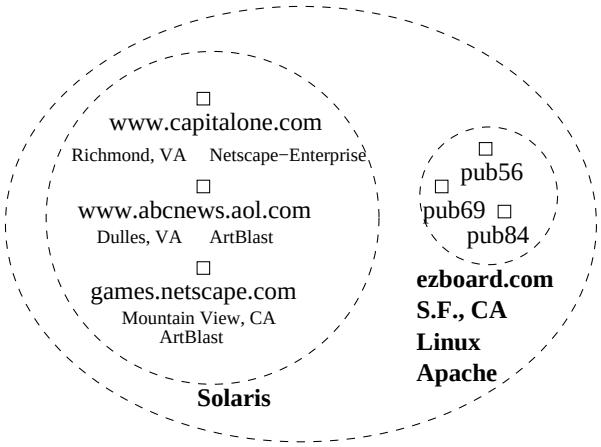

Figure 5. *Two example clusters using the normalized cuts algorithm with mutual information as a metric. One cluster is possibly in the same server room of ezboard.com where all the servers have the same operating system, and web server application. The other cluster is spread more geographically, but all servers use solaris as an operating system and two servers are located in Virginia.*

used three different cluster sizes (16, 32, and 128) since both clustering algorithms take the number of clusters as input. In all, we produced twelve different clusters from the same data input.

Sanity Check In the two weeks that we monitored the 1909 web servers, only 306 exhibited at least one failure. We did not cluster the other 1603 servers because there was no uncertainty in the time that we monitored the nodes; that is, the nodes were always up. Hence, there would be no reduction in uncertainty (i.e. entropy) and the mutual information would be zero. Therefore, we performed the clustering algorithms only on the 306 nodes that showed failure.

Both hierarchical and normalized cuts clustered machines together that are likely to be colocated, on the same powergrid, and/or on the same subnet. Figure 5 shows some of the clusters that the normalized cuts algorithms created, demonstrating that nodes that were likely to fit the above profile were clustered together.

Cluster Algorithm Comparison Hierarchical clustering and normalized cuts have two different objective functions. Hierarchical clustering attempts to minimize the intercluster similarity between clusters in a greedy scheme. Due to the greediness of the algorithm the size of the clusters are often highly skewed (i.e. many nodes in one cluster and many clusters with few nodes). On the other hand, normalized cuts attempt to simultaneously maximize the intracluster similarity and minize the intercluster similarity. This leads to a more even distribution of cluster size, but possibly increases the intercluster similarity.

Days	Hier#16	Norm#16	Hier#32	Norm#32
2	2.1082	2.0976	2.0084	2.0582
3	1.5607	1.5262	1.4702	1.4991
4	1.2789	1.3734	1.3091	1.3432
5	0.9138	1.0844	1.0157	1.0622
6	0.7928	1.0331	0.9631	1.0144

Table 1. *Given a model from the traing set (first week), we computed the average distance (mutual information or joint probability) between a node in one cluster and all the nodes in the other clusters with an increasing number of days in the test set. This table shows the relative average distances for the different clustering algorithms. We used mutual information for this table, but joint probability has similar relative differences.*

In the data we gathered, for hierarchical clustering into 16 clusters, 13 clusters had one node and one cluster had the majority of the other nodes. Hierachical clustering into 32 and 128 clusters were similarly skewed. The normalized cuts into 16, 32, and 128 had more evenly distributed cluster sizes.

We computed the average mutual information (or joint probability) between nodes in different clusters after the first week; that is, the *training set* or "the model". Then we computed the average distance between all nodes not in the same cluster with the second week of data, the *test set*. Table 1 shows the relative comparision between hierarchical and normalized cuts clustering. Notice that hierarchical clustering has a lower average relative to normalized cuts, as mentioned earlier.

9 Discussion and Future Work

Results in the previous section was extremely preliminary. With as little as two weeks of uptime statistics, it is remarkable that obvious clusters can be created demonstrating correlation between sets of nodes. However, some nodes that are likely related may not be clustered with the limited data we collected. Further data gathering and analysis is clearly required.

In addition, more work needs to be done to define an objective function that clusters similar nodes together and generates clusters of reasonable size. There is an interesting tradeoff between the degree of non-correlation between clusters and the freedom to chose dissemination sets (many clusters of equal sizes). For instance, it is easy to put all nodes in a single cluster; this is not particularly useful for dissemination. More than n clusters are desirable during dissemination – should one or more of them be unavailable.

There are many features that observation and analysis may not capture. For example, the same code base (e.g. operating system) managed by different companies may exhibit different failure characteristics and therefore be placed into different clusters. For Byzantine Agreement, we may need to take an extra step and make sure that we pick nodes from different clusters in such a way that the dissemination set has different software base (operating system) and is spread geographically. The analysis of "types" of independence may need to be done on a per-application basis.

In a world-scale system, data gathering and analysis must be a collaborative process. The collective probing of millions or billions of nodes at fine granularity is something that should probably be performed by many individual Model Builders. The resulting data would need to be combined together in a scalable fashion to form a single model. Even more difficult is the question of trust: Set Creators must decide whether to trust models generated by others. Techniques for validating information generated by others are an open problem at this time.

10 Conclusion

In this paper, we present a framework for online discovery of groups of server nodes that are maximally independent in their failure characteristics. We discuss the framework in detail and provide a preliminary evaluation. This type of framework is necessary to achieve the strong reliablity guarantees promised by m of n replication techniques. As peer-to-peer systems become commonplace, techniques for correlation analysis will become increasingly important.

References

[1] Netcraft. http://www.netcraft.com/.

[2] 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.

[3] A. Butte and I. Kohane. Mutual information relevance networks: functional genomic cluster ing using pairwise entropy measurements. In *Proc. of the Pacific Symposium on Biocomputing*, 2000.

[4] 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, Feb. 1996.

[5] 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.

[6] 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.

[7] 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.

[8] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed data location in a dynamic network. In *Proc. of ACM SPAA*, 2002.

[9] S. C. Johnson. Challenges in evaluationg distributed algorithms. In *Psychometrika*, volume 32, pages 241–254, Sept 1967.

[10] I. Keidar. Challenges in evaluationg distributed algorithms. In *Proc. of International Workshop on Future Directions of Distributed Systems*, 2002.

[11] I. Keidar and K. Marzullo. The need for realistic failure models in protocol design. In *Proc. of International Survivability Workshop*, 2002.

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

[13] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM TOPLAS*, 4(3):382–401, 1982.

[14] D. Oppenheimer and D. A. Patterson. Benchmarking large-scale internet services. June 2002. Submitted for publication to Workshop on Dependability Benchmarking.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of SIGCOMM*. ACM, August 2001.

[16] 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, Sept. 2001.

[17] J. Shi and J. Malik. Normalized cuts and image segmentation. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 22, pages 888–905, 2000.

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

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