Survivable Replica Placement in Tree-Based Dependency Models

K. Alex Mills, R. Chandrasekaran, and Neeraj Mittal

Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75080, USA

Abstract

In complex systems, such as data centers, component failure is ubiquitous. Studies have shown that many complex systems do not effectively mitigate the impact of correlated failures. A model for handling correlated failure based upon hierarchical dependencies between multiple points of failure is presented. The dependency model is generic in the sense that it makes no assumptions concerning the hardware or system domain to which it may be applied, therefore it can be applied to model dependencies among failure events in multiple contexts. Under this failure model, the problem of placing replicas on storage nodes is considered. This problem can be solved by a dynamic programming algorithm which computes an optimal placement of a single data replica with respect to a hierarchical failure model in polynomial time. Finally, the limitations of the model are discussed, along with planned improvements.

1 Introduction

Consider a distributed data center in which many blocks of data are stored on thousands of servers. The owners of this data center are being paid both to store data for their customers and to ensure that the data is available whenever their customers require access. In today’s competitive market, customers have strict quality of service requirements for data center providers. These requirements are formalized in a contract, known as a Service Level Agreement (SLA), which states the minimum level of service which must the data center must meet. SLAs typically include agreed upon requirements concerning various performance indicators including system availability and downtime.

Despite financial incentives to meet SLA requirements, many companies still have not implemented best practices for data center management. A 2013 study by The Ponemon Institute [1] surveyed 584 individuals who are directly responsible for data center operations. Only 38% of the respondents agreed that there were ample resources to bring their data center back online in case of an unplanned outage. Around 91% of these individuals reported an unplanned outage, averaging two complete data-center shutdowns over a two-year period. In the presence of such ubiquitous failures, operations managers need to do everything they can to ensure the availability of their customers’ data.

Consumers expect that web services are always available; with the recent surge towards cloud computing, data centers end up bearing this responsibility. In cloud environments, the failure of system components can cause simultaneous failures for multiple web services. Such a scenario happened on July 30th, 2013 in a data center located in Provo, Utah, when two core switches failed during routine network maintenance. The incident took the entire day to diagnose and repair,
which “resulted in several major brands and several million websites knocked out, many of whom probably did not even know that they were customers [of the company who was experiencing the outage]” (emphasis added) [18]. For customers, the financial impact of these failures is significant. In May 2013, Sears filed a lawsuit in which they claim that a profit loss of two million USD due to failures in the power components which serviced their data center. The root cause of this incident involved the simultaneous failure of multiple UPS components on two occasions [12]. In both of these scenarios, multiple co-occurring failures were responsible for the outage.

These examples are not isolated incidents of co-occurring failure. In 2010, Google published results of an extensive one-year study on availability in their cloud storage infrastructure [4]. The study reports that “a large fraction of failures happen in bursts”, indicating that, rather than failing one at a time, servers tend to fail in groups. Furthermore, they found that failing to take such correlated failures into account can lead to an overestimation of availability by several orders of magnitude. They concluded that “correlation among node failures dwarfs all other contributions to unavailability in our production environment”. Google’s analysis confirms the anecdotal evidence of [18, 12]. In both of these scenarios, simultaneous failure events were the ultimate cause of unavailability.

We conjecture that the correlation found among failure events arises due to dependencies among system components. Much effort has been made in the literature to produce quality statistical models of correlation. To our knowledge, all such models ignore the relationship between system dependencies, which can themselves be explicitly modeled since they are known to the system designers. Such dependencies are explicitly represented in our model. Because of this, our system is able to determine the set of nodes which may be affected by a root cause failure. Our model has the potential to characterize and explains highly-correlated failure bursts seen in practice. By including information concerning system dependencies, we can make better informed decisions for data center operations.

To achieve high availability, data centers typically store multiple replicas of data to tolerate the potential failure of system components. This gives rise to the replica placement problem, which, broadly speaking, involves determining which subset of nodes in the system should store a set of data replicas so as to maximize a given objective function (e.g., reliability, communication cost, response time, access time). In practice, the number of replicas to be maintained in the system (referred to as the replication factor) may be either static or dynamic. Variants of replica placement problems have been studied by the computer science community since 1989, as discussed in Section 2.

Our solution solves the replica placement problem in a model which approaches correlation among failures in a novel way. It is based on the observation that root-cause failures in data centers occur independently of one another. Correlation among failure events is induced by dependency relationships among system entities. For example, a data node depends upon its top of rack (ToR) switch for network connectivity. If each of a given rack’s ToR switches fail, the data node will no longer be connected to the network. A similar relationship exists for data nodes with respect to power and cooling subsystems; failure in which may compromise data availability. Our model stands in stark contrast to many statistical models of correlation previously proposed which, while concerning the same problem domain, do not leverage dependency relationships among system components. Rather, they analyze nodes which are time-correlated for failure, and study correlation among the time at which failure events occur. We propose instead a model based upon the causal properties of failure which exist in distributed systems.
2 Related Work

As mentioned previously, correlated failure was known to be a cause for concern as early as 1989, when the first reliability models for batch-job mainframe systems were proposed [8]. Failure propagation models were investigated in the context of distributed systems as early as 1992 [16]. Correlated failure in replicated file storage systems was investigated in [2], where the authors show that correlation can be exploited by storing related files on separate nodes. The idea of exploiting the dependence structure in replica placement is central to our work as well. Distributed algorithms for discovering maximal sets of “independent” nodes were proposed and evaluated in [19]. In 2006, researchers from Carnegie-Mellon, Intel, and Microsoft jointly proposed a set of design guidelines for tolerating correlated failure, justified by qualitative analysis of real-world failure traces [11]. They observed that “selecting between two designs...based on their availability under independent failures may lead to the wrong choice” in design. Furthermore, “additional fragments/replicas result in a strongly diminishing return in availability improvement...”. Our working hypothesis is that these diminishing returns can be increased by an intelligent placement strategy which takes into account causal links for failure, rather than time-dependent correlation.

Regarding correlated failure, our work represents a departure from the traditional approach, which we describe as “measure-and-conquer”. In existing models of correlated failure (i.e. [4, 2, 19, 11]), empirical methods are used to compute the factor of correlation among failures in systems where component dependencies are fully known to the system designers and administrators. In these models, failures are measured via a post-mortem approach in which logs and system history are analyzed to determine the time and location of system failures. Typically, correlation is presumed when two failures occur within a specified unit of time. The downside to this model is that it does not take full advantage of knowledge of the system and its inter-dependencies. It instead treats the system as a black-box, analyzing failure events using only temporal failure information, which may not accurately represent the complex interaction between failure events. Instead, our approach explicitly models the dependency graph which induces the correlation among failure events observed in practice. By explicitly modeling dependency relationships, we believe that better decisions can be made to successfully guard against the impact of failures.

Replica placement has been previously studied in a variety of contexts. Early models for replica replacement came about as extensions of the classical File Assignment Problem, which is surveyed in [3]. Classical models tended to focus on networks in which failure was not considered, with the goal of placing replicas to minimize communication cost, response time, or access time, or to maximize throughput. By the 1990s, new objectives had arisen which sought to ensure reliability under failure. Building on results in [17], the problem of finding a single site in a general graph which maximizes availability under link failures was found to be #P-complete by Johnson et al. in 1994 [6]. This implies that the problem may be harder than any problem in NP. Stephens et al. extended the work in [6] to specific classes of networks including tree and ring networks [14, 15] (for which the problem can be solved efficiently) and also for the case in which more than one replica may be placed.

Other work has focused on finding optimal placements in (undirected) networks under a variety of distributed access models and mutual exclusion protocols. A variety of system assumptions have been analyzed: asynchronous link failures under single-link failure in [9], read-any/write-all protocols in [5], and majority voting protocols in [13, 20]. It is also worth mentioning that the authors of [7] proved an upper-bound on performance obtained by a large class of protocols. In contrast, the work we propose is protocol-agnostic, and concerns optimization of a system regardless of the distributed access technique used.
3 Survivable Replica Placements

To model dependency among system components, we first abstract away the notion of a component and focus instead upon the relationship between distinct, independent failure events. To this end, we define a failure point as any event whose occurrence implies that a set of other failure points are also at risk of failure. Failure points can be used to represent the failure of system components, or even more abstract events which imply complete or partial failure, such as earthquakes or floods.

We represent failure points as nodes in a dependency graph \( G = (V, E) \), a directed graph in which the edges encode information concerning dependency among failure points. Specifically, for two failure points represented by nodes \( u \) and \( v \), an edge \( (u, v) \) exists in \( E \) if and only if the failure of \( u \) implies that \( v \) is also at risk of failure. For the purposes of this report, we make the pessimistic assumption that the failure of \( u \) guarantees the failure of \( v \), (i.e. \( P[v \text{ fails} | u \text{ fails}] = 1 \)). This assumption has the effect that if node \( u \) fails, any node which is reachable from \( u \) in \( G \) also fails. We call nodes which fail because of the failure of an ancestor “dependently failing” nodes, all other nodes fail “independently” (independently failing nodes may be seen as root-cause failures). For the remainder of the report, we will refer to nodes in \( G \) as either “failure points” or “nodes” without making a distinction between the two.

For the purpose of modeling the replica placement problem in the context of data-centers, we restrict ourselves to the case in which \( G \) is a directed tree, with edges directed away from the root. Furthermore, we consider that storage servers are only placed at the leaves of the tree. When the failure point associated with a storage server fails, all replicas placed at the associated server are said to be unavailable. When modeling a data-center, the root node of \( G \) may not have a direct correspondence with any system component. In terms of our application, the root represents a disaster scenario such as an earthquake or flood, during which all data becomes unavailable. We define any dependency graph with these properties as a dependency tree.

In the replica placement problem, we are given a block of data, and \( \rho \in \mathbb{N} \), and we are asked to place \( \rho \) replicas of this data block among storage nodes in a manner which optimizes some criteria. In general, it is undesirable for two replicas of the same block of data to be stored on the same server (this would defeat the purpose of replication). Therefore, with regards to a dependency tree \( T \), we can define a replica placement as a non-empty subset of the set of leaf nodes of \( T \). The size of a placement is naturally defined as the cardinality of this subset.

As regards our optimality criteria, we seek to find a placement which ensures that the number of replicas which survive any given failure is maximized. To this end, given a placement \( P \) of replicas in dependency tree \( T \), we define the survival number of a node \( v \in V \) as the number of failure points in \( P \) which remain available in \( T \) given that \( v \) fails. In other words, it is the number of nodes in \( P \) which are not reachable in \( T \) from \( v \). Under this definition, an independently failing node can be interpreted as detaching itself and all of its successors from the tree. The survival number of a node yields how many replicas from placement \( P \) would remain connected to the root were this to occur. A node is said to be \( \sigma \)-safe if it has survival number greater than or equal to \( \sigma \in \mathbb{N} \); otherwise, it is said to be \( \sigma \)-unsafe.

We consider a solution to the following problem:

**Problem 1.** Given a dependency tree \( T \), a replication factor \( \rho \in \mathbb{N} \), and a survival number \( \sigma \in \mathbb{N} \), find a placement of \( \rho \) replicas in \( T \) such that the number of \( \sigma \)-safe nodes of \( T \) is maximized.

Intuitively, solving this problem ensures that the majority of failure events leave at least \( \sigma \) surviving replicas of a given data-block. In the next section, we solve the problem of placing a single block of data which maximizes our optimality criterion.
4 Survivable Single-Block Placements

We can solve Problem 1 in $O(n^3)$ time using dynamic programming. Dynamic programming is a standard algorithmic technique which solves problems recursively in a “bottom-up” manner. The specific recurrence involved in our algorithm computes an optimal placement for every subtree, and combines these subtrees, starting from the bottom of the tree at the leaves and working its way up the tree towards the root. This section is broken into two subsections. In Section 4.1 the method for traversing the tree bottom-up is described, and in Section 4.2 the dynamic programming recurrences are defined.

4.1 Incrementally Building Trees

Dynamic programming is a technique for solving overlapping subproblems. As in any dynamic program, a key point in our algorithm is the order in which subproblems are solved. In this section we describe this order, emphasizing how the computation is performed. The topic of what is being computed is addressed in Section 4.2.

Given any tree, $T = (V,E)$, we can build $T$ incrementally starting from the leaves. This process can be thought of as visiting edges of $T$ in a certain order. During the algorithm, we maintain a list of subtrees for which the problem has already been solved. As we show in Section 4.2, our problem is trivially solved for a single leaf node, so without loss of generality, suppose this list initially contains all leaves. At each step of the algorithm, we merge two subtrees together by visiting the edge connecting their roots. Specifically, at each edge, we either (a) add a node which has not yet been added to any subtree, or (b) merge two existing subtrees by an edge between their root nodes.

For any subtree rooted at a node $n$, there are two types of edges adjacent to $n$, (1) incoming edges, which connect $n$ to its parent, and are unique to any node, and (2) outgoing edges which connect $n$ to its children. For each node $n$, our technique is only valid when all outgoing edges of $n$ are visited before $n$’s incoming edge is visited. This property is respected by a post-order traversal of tree $T$. Post-order traversal can be described as follows: for each node, recursively traverse all of its child edges, then visit the node’s parent edge (the root node does not have a parent, so in its case, no parent edge is visited, and the algorithm terminates). By visiting edges in post-order, we ensure that we have the results of all child computations available whenever a parent edge needs to be computed. Once all of a node’s child edges have been visited, the output of these computations can be combined to form the output for the node’s parent edge, as we describe in Section 4.2.

![Post-order edge traversal](image-url)
The ideas presented above can be restated as follows. Our algorithm recursively processes subtrees in three cases: the base case, in which a leaf node is processed, and two recursive cases (“up” and “out”) in which we merge nodes by visiting edges. In the “up” case, we add the incoming edge of the root of a subtree, thereby merging the subtree with a node which has not yet been included. In the “out” case, we add an outgoing edge of a subtree, thereby merging two existing subtrees.

An example tree with edges labeled 1-11, is given in Figure 1. In the figure, edges are visited in decreasing order of label. When visiting the edge labeled 4, the subtree consisting of nodes \{e, j, k, l\} has already been recursively computed. Since node b has yet to be visited, edge \((b, e)\) is an “up” edge. Adding edge \((b, e)\) creates subtree \(\{b, e, j, k, l\}\). Compare this to the “out” edge, \((a, d)\), which merges subtrees \(\{a, b, c, e, f, g, j, k, l\}\), and \(\{d, h, i\}\).

4.2 A Dynamic Programming Solution

In this section, we describe what is computed for each subtree, and briefly argue that it maximizes the intended objective. Let \(T\) be a dependency tree, \(\rho\) be the desired replication factor, and \(\sigma\) the desired survival number (see Problem 1). We assume without loss of generality, that \(1 \leq \sigma \leq \rho\), and that \(T\) has at least \(\rho\) leaf nodes; otherwise there is no solution which places \(\rho\) replicas. Let \(U\) denote the set of vertices in the subtree of \(T\) which is currently under consideration, and let \(r(U)\) denote its root. Let \(c(U)\) denote the smaller of \(\rho\) and the number of leaf nodes of \(U\). At most \(c(U)\) replicas can be placed at any subtree \(U\), since we only allow one replica to be placed at any leaf node.

We define \(m(U, k)\) as the maximum number of \(\sigma\)-safe nodes in any placement of \(k\) replicas in \(U\), where \(0 \leq k \leq c(U)\), assuming that \(\rho\) replicas are being placed in the tree overall. We will give a recursive definition of \(m(U, k)\) which can be used to solve the problem, but first we will make some observations concerning the relationship between \(k, \sigma,\) and \(\rho\).

In subtree \(U\), suppose there is a placement \(P\) of \(k\) replicas which guarantees that at most \(m(U, k)\) replicas are \(\sigma\)-safe. Whether \(r(U')\) is \(\sigma\)-safe, can be determined using only the values of \(k, \rho,\) and \(\sigma\). Notice that when \(k \leq \rho - \sigma\), even if \(r(U')\) fails and all \(k\) replicas in \(U'\) become unavailable, there are at least \(\sigma\) replicas stored elsewhere in the tree (see Figure 2). More specifically, if \(\rho - k \geq \sigma\), then \(r(U')\) is \(\sigma\)-safe. Observe that the implication holds in the other direction as well (i.e., if \(r(U')\) is \(\sigma\)-safe, then \(\rho - k \geq \sigma\)).

\[
\begin{align*}
T & \\
U' & \\
\text{\{k replicas\}} & \\
\text{\{\rho replicas\}}
\end{align*}
\]

Figure 2: Depiction of \(\rho - k \geq \sigma\). \(k\) replicas are placed on tree \(U'\), but \(\rho\) replicas are present in \(T\). If all replicas in \(U'\) fail, then \(\rho - k\) replicas survive. If \(\rho - k \geq \sigma\), then \(r(U')\) is \(\sigma\)-safe.
It is possible to recursively compute \( m(U, k) \) by considering the three cases mentioned in Section 4.1 (leaf node, “up” case and “out” case). We consider each case as follows.

1. **Base Case**: In this case, \( U \) consists of a single leaf node of \( T \). For each \( k \) with \( 0 \leq k \leq 1 \):

   \[
   m(U, k) = \begin{cases} 
   1 & : \rho - k \geq \sigma \\
   0 & : \text{otherwise} 
   \end{cases}
   \]

2. **Recursive Case**: As mentioned previously, there are two separate subcases. The first subcase allows a subtree to grow upward by inheriting a parent (“up” case). The second subcase allows a subtree to grow laterally by gaining a child subtree (“out” case). See Figure 3 for illustrations.

   (a) **Up Case**: In this case, the subtree \( U \) is obtained by attaching a child subtree \( V \) to the node \( r(U) \). Notice that merging subtree \( U \) with its parent does not affect the number of \( \sigma \)-safe nodes in \( U \). However, this is not necessarily the case for \( U' \), the subtree formed by \( U \) and its parent. Since \( U' - U = \{r(U')\} \), to calculate \( m(U', k) \) given the value of \( m(U, k) \), we only need to determine whether or not \( r(U') \) is \( \sigma \)-safe, and if so, add its contribution to that of \( m(U, k) \). For each \( k \) with \( 0 \leq k \leq c(U) \):

   \[
   m(U, k) = \begin{cases} 
   m(V, k) + 1 & : \rho - k \geq \sigma \\
   m(V, k) & : \text{otherwise} 
   \end{cases}
   \]

   (b) **Out Case**: In this case, the subtree \( U \) is obtained by attaching a new child subtree \( W \) to the root node of subtree \( r(V) \). Note that \( r(V) = r(U) \). First, we compute an auxiliary function \( \hat{m}(V, k) \) which is identical to \( m(V, k) \) except that it removes any contribution of \( r(V) \) to the value of \( m(V, k) \), when the value of \( k \) causes \( r(V) \) to become \( \sigma \)-unsafe. Therefore, for each \( k \) with \( 0 \leq k \leq c(U) \):

   \[
   \hat{m}(V, k) = \begin{cases} 
   m(V, k) - 1 & : \rho - k \geq \sigma, \\
   m(V, k) & : \text{otherwise.}
   \end{cases}
   \]
Finally, we have, for each $k$ with $0 \leq k \leq c(U)$:

$$m(U, k) = \begin{cases} 
\max_{k=\ell+m} \{ m(V, \ell) + m(W, m) \} & : \rho - k \geq \sigma, \\
\max_{0 \leq \ell \leq c(V), 0 \leq m \leq c(W)} \{ \hat{m}(V, \ell) + m(W, m) \} & : \text{otherwise.}
\end{cases}$$

Note carefully the use of $\hat{m}(V, k)$ in the “otherwise” clause above. In the prior calculation of $m(V, \ell)$, the root of subtree $V$ may have been counted as $\ell$-safe. When $k \leq \rho - \sigma$, the addition of $m$ extra replicas in subtree $W$ can cause this node not to be $k$-safe, therefore the prior contribution of the root to the value of $m(V, \ell)$ must be removed to ensure correct results.

The “out” portion of the recursive case deserves special attention. When computing $m(U, k)$ in the case where an “out” edge is added, two subtrees, $V$ and $W$, are merged to form subtree $U$, with a larger number of leaf nodes on which replicas may be placed. In order to find the maximum number of $\sigma$-safe nodes which may be possible at $U$, we may reuse the result of computations performed at $V$ and $W$. To ensure the maximum is returned, we consider all possible ways to split up $k$ replicas among subtrees $V$ and $W$, and return the maximum.

The maximum number of $\sigma$-safe nodes in any placement of $T$ is given by the value of $m(T, \rho)$. The actual placement can be obtained by carefully keeping track of the values of $\ell$ and $m$ which yield the maximum sum in each of the “out” cases. A technique for keeping track of these values is given in [10].

**Theorem 1.** The algorithm described above solves the replica placement problem in Problem 1 for a given value of $\rho$ and $\sigma$ with $0 < \sigma \leq \rho$. The algorithm runs in $O(\rho^2 |V|)$ time.

**Proof.** The correctness of the algorithm can be easily seen by careful examination of the cases involved, and by applying induction to the structure of the tree. Therefore, we focus on the worst-case running time of the algorithm.

The algorithm visits edges in post-order, merges the subtrees at the vertices adjacent to the edges to form subtree $U$, and computes $m(U, k)$. This occurs once at every edge for a total of $|E| = |V| - 1$ edge visits. At each edge, we must compute $m(U, k)$, for $0 \leq k \leq \rho$. The bottleneck in the computation of $m(U, k)$ occurs in the “out” case, where each $k$ must be broken into $O(k)$ compositions of two integers, over which the maximum value is taken. Substituting the upper-bound $k \leq \rho$ yields the result.

5 Conclusion

In this technical report, we have described our dependency model and indicated how to find a survivable replica placement in polynomial time. In the model which we described, nodes can be interpreted as failing with uniform probability. As part of our future work, we plan to extend the model to handle the case in which nodes may fail with distinct probabilities, identify new objective functions for optimal replica placements, and analyze the complexity of optimizing the chosen objectives. Additional future work entails extending our models to handle directed acyclic graphs, and graphs with cycles, as well as analyzing the complexity of replica placement in such models.
References


