Towards Optimal Resource Allocation in Partial-Fault Tolerant Applications

Nikhil Bansal*, Ranjita Bhagwan†, Navendu Jain‡, Yoonho Park*, Deepak Turaga*, Chitra Venkatramani*

*IBM T. J. Watson Research Center †Microsoft Research ‡University of Texas at Austin

{nikhil,yoonho,turaga,chitrav}@us.ibm.com bhagwan@microsoft.com nav@cs.utexas.edu

***Abstract*—We introduce Zen, a new resource allocation framework that assigns application components to node clusters to achieve high availability for *partial-fault tolerant* (PFT) applications. These applications have the characteristic that under partial failures, they can still produce useful output though the output quality may be reduced. Thus, the primary goal of resource allocation for PFT applications is to prevent, delay, or minimize the impact of failures on the application output quality. This paper is the first to approach this resource allocation problem from a theoretical perspective, and obtains a series of results regarding component assignments that provide the highest service availability under the constraints imposed by the application data flow graph and the hosting clusters.**

We show that (1) even simple versions of this resource allocation problem are NP-Hard, (2) a 2-approximate polynomial-time algorithm works for tree topologies, and (3) a simple greedy component placement performs well in practice for general application topologies. We implement a system prototype to study the application availability achieved by Zen compared to failure-oblivious placement, replication, and Zen+replication. Our experimental results show that three PFT applications achieve significant data output quality and availability benefits using Zen.

I. Introduction

With increasing scale and complexity of deployed distributed applications, their utility is increasingly limited by availability rather than performance [11], [32]. Thus, masking failures to ensure application availability has become a key goal in dependable computing. Traditional approaches to fault-tolerance are based on techniques such as replication [4], [13], [17] and checkpointing [13], [14], [20]. Multiple replicas increase the probability that an application will be able to provide service to end-users despite individual node failures. Similarly, checkpointing allows a service to be restarted quickly from a backup site, thereby limiting the work lost when a failure occurs.

However, these approaches introduce well-known tradeoffs between cost and availability. For example, a replicated service may incur significant overheads to provide strict consistency requirements [13], [33]. Further, the monetary cost of implementing highly available services can double for just a fraction of percentage of availability [8], and under correlated failures, even additional replicas result in a strong diminishing return in availability improvement for many replication schemes [21]. Similarly, the overheads of checkpointing can limit its benefits [34].

In this paper, we investigate an alternative resource allocation framework whose main philosophy is that if we can leverage knowledge of failure characteristics and resource capacity constraints (e.g., CPU, MEM, I/O, etc.) of node clusters in *component placement* (i.e., to which cluster should each component be assigned), we can achieve significant gains in availability for an important class of applications, which we term *partial-fault tolerant* (PFT) applications. In contrast to applications that require the availability of all components to operate correctly, PFT applications provide a "graceful degradation" in performance as the number of failures increases. For example, aggregation systems such as Sawzall [23], SDIMS [30], and PIER [12] are likely to be able to tolerate some missing objects while processing a query (e.g., JOIN, MEDIAN, AVG, etc.) on a distributed database. Similarly, data mining applications such as WTTW [28] and FAB [27] (described in Section II-A) can still classify data objects under failures, though with less confidence. Further, for many data stream applications with stringent temporal requirements [2], [3], it is more important to produce partial results within a given time bound than full results produced late. Finally, mission-critical applications (e.g., power control, telecom, medical systems) deploy multiple sensors [9] such that at least some of them should be able to trigger an alert during failures or when operating conditions are violated (e.g., when a sensor senses a fire.)

Given this system model, we can precisely state the problem as follows: Given a distributed computing system comprising n clusters $(T_1, T_2, \ldots, T_n)$ each with a resource capacity c_i and a failure probability p_i $(i \in [1, n])$, and a PFT application made up of m components $(C_1, C_2, \ldots, C_m)$ each of which may execute on any cluster, allocate each of the m modules to one of the n clusters such that the *loss in expected application*

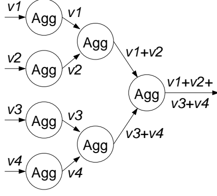

Fig. 1: Data aggregation application.

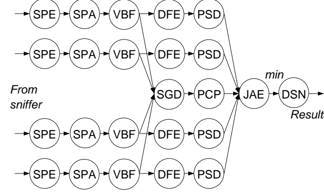

Fig. 2: WTTW data flow graph.

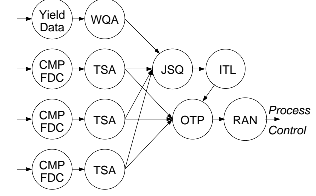

Fig. 3: FAB data flow graph.

output value is minimized under failures subject to the constraints imposed by the application data flow graph, the resource capacities, and the failure probabilities.

To solve this problem, we introduce the Zen framework that performs *failure-aware component assignment* to achieve high availability for PFT applications. Zen approaches the problem from a theoretical perspective by reducing the above optimization problem to a variant of the graph coloring problem. We show that even simple versions of this graph coloring problem are NP-Hard. Therefore, Zen provides (1) a 2-approximate (wrt the optimal algorithm's output loss) polynomial-time algorithm for tree topologies and (2) a greedy algorithm using the underlying principles of the optimal algorithm for general graph topologies, and shows that it works well in practice.

We have implemented a prototype of Zen and evaluated its effectiveness by deploying across a cluster testbed three real-world PFT applications with a broad range of scale, topology, functionality, and fault-tolerance a data aggregation application, WTTW, and FAB. Relative to a failure-oblivious placement, Zen improves availability of these applications by 35%-60%. Compared to replication, (1) Zen provides nearly equal availability as replication but with less cost and (2) Zen+replication achieves significantly higher availability from 33%-90% for the same cost as replication.

By using Zen, we hope that application managers can significantly improve service availability by making an *application-aware* and *cluster-aware* automated resource allocation. Our long-term goal is to enable self-adaptive component placement for achieving high availability in large-scale, dynamic environments. Towards this goal, this paper makes three technical contributions:

- We introduce the problem of failure-aware component assignment for PFT applications and show that even simple versions of this problem are NP-Hard.
- We develop the Zen resource allocation framework that provides a 2-approximate algorithm for tree topologies and a greedy algorithm for general application topologies that works well in practice.
- We implement a prototype of Zen in a large-scale stream processing system and evaluate its effect on availability compared to failure-oblivious placement, replication, and Zen+replication. Our experimental results show that three PFT applications achieve significant availability benefits using Zen.

The rest of this paper is structured as follows. In Section II, we discuss motivating applications and examples. Section III defines our system model in terms of the class of failures and applications we target. Section IV gives a proof of the NP-hardness and describes our approximation algorithms. Section V describes evaluation of Zen using our system prototype. Finally, Section VI discusses related work and Section VII concludes.

II. Motivation

A. Motivating Applications

Our work is motivated by PFT applications. In this section, we describe three PFT applications that span a broad range of size, topology, functionality, and fault-tolerance. For each application, we focus on quantifying its availability in terms of the application output quality.

Data Aggregation. Our first application is a data aggregation service that is designed to be representative of in-network query processing systems such as Sawzall [23] and SDIMS [30]. In our implementation, an aggregation tree is constructed (Figure 1) that assigns aggregator components at internal nodes which receive children inputs and send the aggregate output to their parent. An aggregator's data output value is well-expressed as a summary of its incoming data values. Several standard aggregation functions such as SUM, AVG, MAX, MIN, etc. satisfy this application model. Availability in this application is measured by the percentage of paths from the leaves to the root that contribute to the global aggregate. E.g., a COUNT query can tolerate some missing data inputs to count the number of nodes in a network.

WTTW. The Who's Talking to Whom (WTTW) application is a VoIP-based stream-processing application [28]. Its aim is to identify and track conversing parties in real-time from distributed and noisy/compressed speech signals. The data flow graph of WTTW (Figure 2) shows the inputs from various remote sniffers, each carrying several compressed speech signals, that feed data to different operators for processing and classifying conversation pairs. These operators include fast feature extraction

from compressed speech signals (DFE), fast speaker detection in quantized feature space (PSD), progressive stream pairing of conversing parties (PCP) using stream volumetric information (SGD), windowed join (JAE), and iterative denoising of detection events (DSN). The details of these operators are beyond the scope of this paper and are described in [28]. In WTTW, all components' output values are a SUM of their inputs except JAE that performs a JOIN operation. WTTW's quality metric (described in Section V) is defined in terms of the accuracy in correctly identifying conversation pairs [28]. If the output of PCP, which is one of JAE's inputs, goes to 0, the JAE join operation fails entirely. Hence, JAE uses the MIN operator to characterize its output value. **FAB.** The third PFT application, FAB, monitors processing of silicon wafers in a semiconductor manufacturing environment [27]. Its goal is to predict wafer yield from summary vectors input from the tool sensors and detect any tool anomalies in real-time. FAB does self-learning as it continuously evaluates its prediction performance against the ground truth i.e., actual wafer yield. Figure 3 shows data flow graph of FAB; details of FAB operators are described in [27]. the input summary vectors and the ground truth are fed to the OTP classifier that predicts wafer yield and sends it to RAN and JPQ; JPQ then does JOIN with the actual yield. Next, YPV generates a confidence score to evaluate classifier's performance. Finally, ITL receives the outputs of JSQ join and YPV to build a new classifier. FAB's quality metric (described in Section V) is defined as its prediction accuracy that depends on the path availability from the sensors to the OTP classifier and the incremental learner ITL.

B. Motivating example

A simple yet subtle example. Consider the PFT application in Figure 4: component 1 computes a SUM (say) over outputs of components 2 and 3. Figure 4 shows three possible component assignments: α, β, and γ. In (α), we assign root (C_1) to one cluster (black) and C_2 and C_3 to another cluster (gray). In (β), we assign C_1 and C_3 to the gray cluster and C_2 to the black cluster, and in (γ), we assign C_1, C_2, and C_3 to the gray cluster.

Note that allocation β is better[1] than α because if the black cluster fails, then the application output for α goes to 0. However, under allocation β, the system could still process data flowing from C_3 to C_1. Indeed, if the gray cluster fails, both allocations give no output. A careful

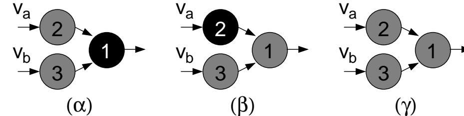

Fig. 4: Three possible assignments of components to 2 clusters.

calculation[2] shows that the best allocation, however, is γ that keeps all components on the same cluster. The main intuition behind this is that only one cluster failure affects allocation γ while two cluster failures can hinder the other allocations.

There are several important observations from this example. First, allocate as many components as possible to the same cluster (subject to cluster resource constraints) to maximize expected output value under failures. Second, assign components on independent paths to different clusters to avoid dependent failures. Finally, for heterogeneous clusters with different failure probabilities, assign "highly important" components to clusters with the lowest failure probabilities. We use these observations in designing our component placement algorithms in Section IV.

III. System model

In this section, we present the failure and application models based on which we formulate the optimization problem of failure-aware component allocation.

A. Failure model

While many previous studies assume failures to be independent, in reality, the assumption of failure independence is rarely true. Node failures are often correlated, with multiple nodes in the system failing nearly simultaneously [10], [21], [24], [25]. E.g., nodes on a rack connected by a switch or a power supply, network-attached disks, nodes running the same version of a vulnerable software [17], etc. Further, the size of correlated failures can be quite large and have a dominant affect on system availability [21]. We therefore use the model proposed by Junqueira et al. [18] which groups all resources that can fail together into clusters where each cluster can fail-stop in its entirety, causing all its resources and hosted application components to be unavailable. Note that a single cluster failure also affects other components that depend on the output from its failed components. Individual cluster failures are considered independent and identically distributed. Each cluster[3] T_i has a failure probability p_i.

This model is more practical as it represents a number of large-scale infrastructures such as (1) large machine clusters comprising of multiple blade centers (a blade

[1] Assuming a uniform cluster failure probability p, the failure probabilities (i.e., no output from the root) are $FP(\alpha) = p + (1-p)p$, $FP(\beta) = p$, and $FP(\gamma) = p$; $FP(\beta) = FP(\gamma) \leq FP(\alpha)$.

[2] The expected output values are $E[V]_\gamma = (V_a + V_b)(1-p)$ and $E[V]_\beta = V_b(1-p)p + (V_a + V_b)(1-p)^2$; $E[V]_\gamma \geq E[V]_\beta$.

[3] Nodes failing independently can be mapped to clusters of size one.

center is a "cluster") connected by a switch, and therefore may get disconnected due to link failures, switch misconfigurations or localized power outages and (2) wide-area distributed systems that may get network partitioned due to network disconnections, software errors [31], system misconfigurations, DoS attacks, worms and viruses [17], etc. and (3) distributed data stream applications model applies to stream-processing applications in a slightly different way. In a stream processing system, an application is divided into a set of components which the system allocates to different machines. When a machine fails, it effectively causes the failure of all the application components that run on it. The application components are the “processes” and each machine is the “cluster” referred to in the failure model.

Given a placement of the components onto the clusters, if some cluster fails, then all the components that are allocated to this cluster fail. Failure of a component has a cascade effect. If a component fails, then the value of the data produced by it goes down to 0, which affects the value of this component’s parent in the tree, and so on.

B. Application model

In this work, we mainly target distributed data-processing applications consisting of multiple networked components. A PFT application topology is modeled as a directed graph of inter-connected components where vertices are either data sources, sinks, or processing operators, and edges represent the data flow between them. We assume each input source s (e.g., a leaf in a tree) having a certain *importance value* v_s (scalar value) defined as s’s contribution to the application output. Alternatively, v_s is the “loss” incurred in the application’s total output value if s fails. The importance value of a component C (output) is expressed as a linear combination[4] or a MIN/MAX of its input importance values. Intuitively, each C performs “value addition” to its data inputs that can be modeled as a linear function in many applications. E.g., for a tree topology, v_i of an internal node i is conceptually a SUM of v_l for all leaves l lying in the subtree rooted at i. Further, this simple model fits well the real applications described in Section II-A. For more complex applications, we require application designers to specify the relative importance values of the components. The importance metric allows us to rank application components so that highly important components get assigned to highly reliable clusters.

[4] $\alpha_{e_1}v_1 + \alpha_{e_2}v_2 + \ldots + \alpha_{e_k}v_k$ where α_{e_i} is the weight associated with edge e_i receiving input v_i from the ith child of C.

IV. Problem Hardness and Algorithms

As defined precisely in Section I, given a PFT application A, our problem is to assign A’s components to a set T of clusters, each cluster T_j with capacity c_j and failure probability p_j, such that the loss in the expected application output is minimized.

A. Overview

irst, we show that even simple cases of the component assignment problem (Section I) are NP-Hard. Second, for applications with tree topologies, we provide a polynomial-time optimal algorithm assuming an unbounded number of homogeneous clusters ($c_i = c, p_i = p\ \forall i$), and a 2-approximate (wrt optimal’s output loss) algorithm under a bounded number of clusters. Third, for general graph topologies, we present a greedy algorithm using the underlying principles of the optimal algorithm and show that it works well in practice in Section V.

B. Optimal Component Allocation is NP-Hard

To prove optimal component allocation is NP-Hard, consider a simple tree-based PFT application (e.g., Figure 1) computing an aggregation function. For this application, we first characterize the effect of cluster failures on the output value using the following lemma:

Lemma 4.1: Let $X : P \to N$ be some fixed placement of components to clusters. For a leaf l, let $P(l)$ be the set of components on the path from l to the root, and let $S(l)$ denote the set of clusters that have at least one component in $P(l)$ assigned to them. Let v_l denote the importance value of the input entering l. If p_j denotes the probability of failure of cluster j, then the expected output value of the application is

$$\sum_{l \in leaves} \left(\Pi_{j \in S(l)}(1 - p_j)\right) v_l$$

If all the failure probabilities are equal i.e., $p_j = p\ \forall j$, then the expected output value becomes

$$\sum_{l \in \text{leaves}} (1-p)^{d(l)} v_l$$

where $d(l) = |S(l)|$ is the number of distinct clusters on the path $P(l)$.

Proof: Let us consider input data that enters a leaf l and follows the unique path $P(l)$ from l to the root. If no cluster (and hence component) on this path fails, then this stream contributes a value v_l to the root aggregate value. Otherwise, if any component fails on $P(l)$, this input contributes 0. The probability that no component on the path $P(l)$ fails is exactly the probability that no cluster in $S(l)$ fails, equal to $\mathrm{pr}(l) = \Pi_{j \in S(l)}(1 - p_j)$ since cluster failures are independent. Thus the expected contribution of leaf l to the root output is $\mathrm{pr}(l) * v_l$. By

linearity of expectation, the expected root output value is $\sum_{l\in\text{leaves}} \text{pr}(l)v_l$. If $p_j = p\ \forall j$, then $\Pi_{j\in S(l)}(1-p_j)$ = $(1-p)^{|S_l|} = (1-p)^{d(l)}$ by definition of $d(l)$. ■

Given Lemma 4.1, we can view the component assignment problem as the following tree coloring problem: Given a rooted tree $T = (V, E)$ with vertex set V and edge set E, assigning components optimally to a set of k clusters of capacity $c_j (j \in [1, k])$ is equivalent to coloring V with colors $1, 2, \ldots, k$ such that color j is used no more than c_j times and the term $\sum_{l\in\text{leaves}}(1-p)^{d(l)}v_l$ is maximized, where $d(l)$ is the number of distinct colors on the path from leaf l to the root.

Intuitively, the goal is to color the tree vertices such that each path has as few colors as possible. Translating back, we want to minimize the total number of distinct clusters used on every path from a leaf to the root. Observe that if no cluster fails, the maximum achievable output value is $\sum_{l\in\text{leaves}} v_l$. Thus, we can define the *loss* of a placement to be the difference between the maximum achievable value and the value achieved by the placement i.e., loss $= \sum_{l\in\text{leaves}} v_l(1-(1-p)^{d(l)})$. Hence, maximizing the expected output value is equivalent to minimizing the loss[5]. Taking the first order approximation, our objective function of minimizing the loss in the output value becomes:
Min $\sum_{l\in\text{leaves}}(1-(1-p)^{d(l)})v_l \approx$ Min$\sum_{l\in\text{leaves}} pd(l)v_l$

This has the advantage of linearizing the problem and making it tractable.

Now, we can define the loss minimization problem as:

Definition 4.2 (Loss minimization problem): Given a tree $T(V, E)$ with input values v_l at leaf l, color V from colors $1, 2, \ldots, k$ such that color i is used at most c_i times and the term $\sum_l v_l d(l)$ is minimized where $d(l)$ is the number of distinct colors on the path from l to root.

Notice that p has been dropped from the loss definition since it is a fixed constant. and minimizing the loss as defined above is equivalent to that with p multiplied.

Figure 5 gives an example graph that can represent an aggregation tree-based application. It shows two possible colorings with cluster capacity c set to 3. In allocation (a) we see that all paths from leaf to root have exactly two colors. On the other hand, allocation (b) has two colors on all paths except 7-3-1, which has only one color. Hence, if $v_l = V$ for all components, the value of loss for allocation (a) is $8V$ while for (b) it is $7V$, showing that (b) is a better allocation strategy than (a).

[5]However in terms of approximation guarantees, the two quantities are different. E.g., if an optimal solution achieves 99% of the maximum achievable value, then a 2-approximate algorithm wrt value will only guarantee at least 49.5% value, while a 2-approximation wrt loss will guarantee at least 98% value. Zen provides 2-approximation wrt loss.

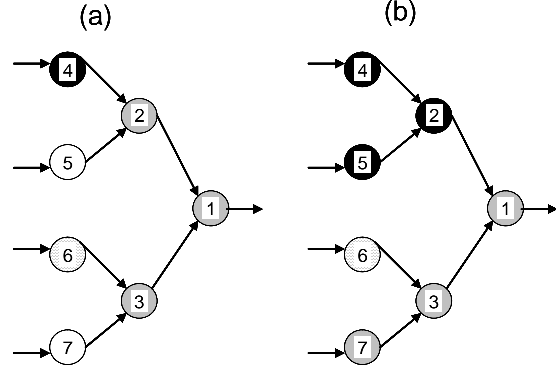

Fig. 5: Two example allocations of a distributed tree.

As the following lemma shows, the loss minimization problem is NP-Hard even for very simple instances.

Lemma 4.3: The loss minimization problem is NP-Hard even for tree instances with maximum degree 3 and all colors (clusters) have equal capacities.

Proof: We sketch the proof by showing a reduction from the 3-partition problem: Given $3m$ non-negative numbers $a_1, \ldots, a_{3m}$, is there a partition of these numbers into m sets such that the sum of each set is equal to S/m where $S = \sum_{i=1}^{3m} a_i$? This problem is NP-Hard even when S is polynomially large in m. The main idea of the reduction is as follows: Consider a tree with $3m$ disjoint paths of length $a_1, a_2, \ldots, a_{3m}$, all connected to a common root, and m clusters each with capacity S/m. Since our goal is to minimize the number of distinct clusters on each path, it can be shown that there exists a placement in which each path lies in a single cluster iff the 3-Partition problem has a solution [5]. ■

It is important to note that the reason for NP-hardness does *not* directly result from coloring paths with fewest colors. Rather, it arises due to as we need to carefully partition the components among the limited number of colors available as we next show a polynomial-time optimal algorithm assuming an unbounded number of colors[6] and adapt this solution to obtain a 2-approximate algorithm when the number of colors is bounded.

C. Optimal Component Allocation Algorithm

First, we present a simple lemma that provides a useful structural property for designing the optimal tree coloring algorithm. Consider an optimum solution OPT. We can assume that OPT satisfies the following property without loss of generality.

Lemma 4.4 (Consecutive Vertex Color Property): Consider a vertex v and another vertex v' such that v' lies in the subtree rooted at v. If v and v' are both

[6]Note that unbounded number of colors doesn't make the problem trivial since we still need to color the paths with least number of colors.

assigned color Q, then every component on the path between v and v' must also be assigned the color Q.

Proof: We will show that any solution can be modified in such a way that this property is satisfied without worsening the value of the solution. (the reader might wish to look an the example in the Appendix explaining this). Consider a solution that does not satisfy this property. Then, there must exist a vertex v and another vertex v' in the subtree rooted at v such that both have color Q, but the child of v does not have color Q. Call this vertex v'' and let its color be Q''. Now, consider the solution obtained by swapping the colors of v' and v''. Note that this swap can only possibly affect the value $d(l)$ (and hence the contribution to the overall solution) of the leaves that lie in the subtree rooted at v''. Let L' be the set of leaves that lie in subtree rooted at v' (and hence also v'') and let L'' denote the leaves that lie in the subtree rooted at v'' but do not lie in the subtree rooted at v'. Note that for leaves in L', the colors along the path to root do not change, hence $d(l)$ remains unchanged for these leaves. For the leaves in L'', the color Q'' is replaced by Q, however since v was already colored Q, and as each leaf to root path for leaves in L'' also passes through v, the quantity $d(l)$ can only decrease for each such leaf. Thus the result follows. ■

The above lemma allows us to obtain a polynomial-time algorithm for the coloring problem with infinitely many clusters, which we shall now explain.

Algorithm sketch. Consider a tree T under OPT color assignment. By Lemma 4.4, all nodes colored i form a connected subgraph. Let r' be a node colored i and let T' denote the subtree rooted at r'. Consider the trees T' and $T'' = T \setminus T'$. Since we want to color paths with the least number of colors, only color i can be shared[7] between T' and T'', and hence the only connection between the solutions for T' and T''. We use this observation in a dynamic programming formulation [5]o compute $V(T', q)$ starting bottom-up from the leaves where $V(T', q)$ denotes the optimum solution for T' under the constraint that the color of T''s root is used at most q times. Clearly, computing OPT corresponds to computing $V(T, c)$ where any color can be used at most c times.

Next, we show how to compute $V(T', q)$ bottom-up: Suppose we color the root r by color 0. Let $T_1, \ldots, T_k$ be the subtrees rooted at children of r. Using Lemma 4.4, we know that for a subtree T_i, either its root is also colored 0, or the color 0 is never used in the subtree T_i.

[7]Having an unbounded number of colors enables this sharing since we are not constrained by the *total* capacity.

Case 1: If a subtree T_i does not use color 0, we claim that the loss that T_i contributes to T is $\text{loss}(T_i) + \sum_{l \in L_i} v_l$. Here L_i is the set of leaves in T_i and $\text{loss}(T_i)$ is the minimum achievable loss for T_i. This holds because the subtree T_i does not contain the color 0, but the parent of the subtree, r, is colored 0. Hence for every path from a leaf in T_i to the root, the value of $d(l)$ (i.e., the number of colors from a leaf to the root) increases by exactly 1. Since the second term ($\sum_{l \in L_i} v_l$) is a constant, it suffices to find the best solution to T_i subject to the constraint that it does not contain color 0.

Case 2: If the subtree T_i uses color 0, the root of T_i must be colored 0. For every leaf in L_i, since each path to the root of T_i already has color 0, the number of distinct leaf to root colors $d(l)$ in T is equal to the number of distinct colors from the leaf to the root of T_i. Thus, the loss contribution of T_i to T is $\text{loss}(T_i)$. Hence it suffices to compute the best solution to T_i subject to the constraint that the root of T_i is colored 0.

We use these observations to design a polynomial-time dynamic programming based optimal algorithm [5] for the loss minimization problem. The optimal solution runs in time $O(|V|^2 c^2)$ and space $O(|V|^2 c)$.

D. 2-Approximate Algorithm

We next show a 2-approximate algorithm wrt loss to OPT under a bounded number of resource clusters. By bounded resources we mean that we do not have infinitely many colors at our disposal, but only a limited number of colors, each with capacity c. In particular, we can assume the worst possible scenario that the total color (cluster) capacity available is equal to the number of vertices in the tree.

Lemma 4.5: There is a 2-approximation with a bounded number of colors (clusters.)

Proof: Compute the optimal solution as described above using unbounded resources. Note that no color is used more than c times in this solution. Moreover this solution is clearly a lower bound on the optimum achievable loss assuming bounded resources. Let m_i denote the number of times color i is used in the unbounded solution. We have $\sum_i m_i = m$, where $m = |V|$ and each $m_i \leq c$; note that $m \leq c * n$ must hold for a feasible solution. We line the m_i-colored nodes up together one after another, and form blocks of size c. Note that since each $m_i \leq c$, any set m_i-colored nodes can lie in at most two blocks. Each block corresponds to a color in the new solution. In any leaf to root path in the new solution, the number of distinct colors used is at most twice the number of colors used in the original unbounded solution. Thus the result follows. ■

E. Greedy Component Allocation Algorithm

Finally, we present a greedy component placement for general graph topologies under heterogeneous clusters and components.

Given an application component C, we denote its importance as $i(C)$ e.g., $i(C) = \sum_{leaf(l) \in Subtree(C)} v_l$.

E.g., a failure of a leaf l in a SUM aggregation tree incurs a loss of v_l in the output, whereas a root failure incurs loss of $\sum_{l \in leaves} v_l$.

In other words, the more the value loss caused by the failure of a component, the more important it is.

Note that $i(c)$ depends on both c's functionality (e.g., an internal node computing partial SUM) and application topology (e.g., SGD in WTTW.)

Thus, given a PFT application, Using the importance metric (Section III-B), the loss minimization term $\sum_{l \in \text{leaves}} v(l) d_l$ (Section IV-B) can be equivalently expressed as $\sum_{C \in \text{components}} I(C)Z(C)$ where $I(C)$ is the importance of component C, and $Z(C)$ is 0 if C and its parent are placed on the same cluster and 1 otherwise [5]. Thus, to minimize loss, set $Z(C) = 0$ for as many adjacent components as possible, especially the ones with the higher values of $I(C)$.

This formulation suggests two guiding principles consistent with our observations in Section II-B: (1) components of higher importance should be placed on clusters with higher capacities (and low failure probabilities) i.e., choose cluster j with highest $\frac{c_j}{p_j}$ and (2) all components lying on a path from a source to the sink should be co-located on the same cluster (if possible) i.e., minimize the total number of colors (clusters) on all paths.

Consider the two applications described in Section **??**. In the first example, importance is mainly affected by application topology, so the root component is more important than a leaf component. In the second WTTW application, the functionality of the component primarily determines its importance. For example, the SGD, PCP, JAE and DSN components are equally important because the loss of any one of these components causes complete output loss. This is because the JAE's output is defined as a min of its inputs meaning that the failure of any of its inputs can cause it to produce no output. Hence SGD is more important than the PSD components even though the PSD components are closer to the final output. Through these examples we see that the importance of a component depends on both the functionality of the component and the topology of the application.

The idea of minimizing loss hinges on the components' functionality as well as the topology of the application. If data of high value travels on a certain path in the application, then this path must have as few colors as possible, i.e. the components on this path must be placed on as few clusters as possible.

Finally, to model the correlated nature of cluster failures, consider the case when the failures of some clusters are perfectly correlated, i.e. they all fail together or else they all survive. In this case, we can just merge these clusters and view them as one single cluster. In reality, the behavior lies some between being completely independent and being completely correlated. Thus instead of merging them completely, we can merge them partially.

F. Heuristic solution

In this section, we use the intuition obtained from solving the loss minimization problem, lemma 4.1 and lemma 4.4, to propose a heuristic that will solve the problem for the more generalized problem.

We relax a number of the assumptions made in Section **??**. All clusters need not have equal capacity, and components can have differing resource requirements. A component output can be either the sum of the input components, or the minimum value of all input components, as explained in Section **??**. The graph can be an arbitrary, directed acyclic graph.

Figure 2 shows the WTTW application graph which fits this model. The allocation problem in this case is to place the application components on a set of machines, hence each *process* in our failure model is an application component, and each cluster is the individual machine on which these components run. One machine failure is equivalent to the simultaneous failure of all application components running on it.

All the components in this example are additive, i.e. their output value is the sum of their input values, except JAE. This component executes a *join* on the data coming from the PSD components with the data coming from the PCP component. If the PCP component dies, then JAE produces no output. Hence we use the `min` component to determine the value of the output of JAE.

The heuristic uses the following steps: Given an application data flow graph $G(V, E)$, Algorithm 1 allocates components in decreasing importance to clusters ranked by $\frac{c_j}{p_j}(j \in [1, n])$. It defines a connected subgraph SG of components that are co-located on the same cluster (say T) as follows: at each step, assign the highest importance C_k to T (if spare capacity) if $(C_k, SG) \in E$ i.e., $(C_k, C_p) \in E$ for some $C_p \in SG$.

Figure 6 shows how the heuristic works on the WTTW data flow graph. The objective here is to allocate WTTW's components to individual machines. We predetermine the resource usage (in terms of CPU usage, since all our machines are identical) for each of the

Algorithm 1 Greedy Component Placement Algorithm

1: Calculate $I(\boldsymbol{C})$ for components $\boldsymbol{C} = \{C_1, C_2, \ldots, C_m\}$.
2: Rank the clusters $T_1, T_2, \ldots, T_n$ sorted (decreasing order) on $\frac{c_j}{p_j}$ ($j \in [1, n]$).
3: $j \leftarrow 1$
4: **while** $\boldsymbol{C} \neq \phi$ **do**
5: Select the highest importance component $C_i \in \boldsymbol{C}$
6: **while** T_j has spare capacity **do**
7: Assign $T_j \leftarrow C_i$; $\boldsymbol{C} \leftarrow \boldsymbol{C} \setminus \{C_i\}$; $SG \leftarrow \{C_i\}$
8: Select highest importance $C_k \in \boldsymbol{C}$ s.t.$(C_k, SG) \in E$
9: **if** $\exists C_k$ satisfying **(8 :)** AND T_j has capacity **then**
10: $T_j \leftarrow C_k$; $\boldsymbol{C} \leftarrow \boldsymbol{C} \setminus \{C_k\}$; $SG \leftarrow SG \cup \{C_k\}$
11: **else** {no such C_k exists OR T_j has no capacity left}
12: break;
13: **end if**
14: **end while**
15: **if** T_j has no spare capacity **then**
16: $j \leftarrow j + 1$
17: **end if**
18: **end while**

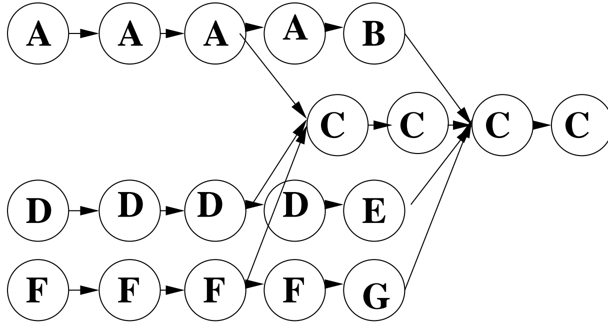

Fig. 6: Heuristic-based placement of the components of the WTTW application.

components and use this to drive step 4 of the heuristic. The components named DSN, JAE, PCP and SGD have the highest importance (as calculated in step 1) since if any of these fail, the output value of the application goes to 0. The rest of the components are relatively less important since, assuming $v_1 = v_2 = v_3 = v$, any of their failures results in the output going down by a third. For example, if one of the PSD components fail, the JAE output reduces to $Min(a = 3v, b = 2v) = 2v$.

For this example graph, each PSD component is resource-intensive and requires the exclusive use of one machine. The rest of the components in the same path are relatively light-weight, causing the heuristic to place them on the same machine. Thus the heuristic makes an effort to keep the number of colors per path as low as possible.

V. Experimental Evaluation

In this section, we evaluate Zen's availability benefits for three real-world PFT applications (Section II-A) compared to failure-oblivious placement (FOP), replication, and replication+Zen. For replication, we create one replica for the highest importance components of each application and assign components to clusters randomly. FOP uses a simplified version of our topology-aware approach by co-locating components in a path on the same cluster but without considering failure probabilities.

A. Methodology

The Zen prototype and the hosted PFT applications run across a cluster testbed on top of the Stream Processing Core [15]. The cluster testbed is an IBM BladeCenter comprising 100 nodes with Intel Xeon and AMD Opteron processors with 2 GB to 4 GB RAM running Linux kernel 2.6.21 connected by 1 Gbit Ethernet. The BladeCenter is arranged into networked chasses where each chassis has up to 13 blades (nodes) and denotes a dependent failure unit (Section III-A) that can fail due to switch misconfigurations, power failures, etc. **Failure Model.** Based on prior studies, we use two failure models: (1) FM1 representing a controlled stable enterprise system and (2) FM2 a dynamic wide-area network e.g., PlanetLab [1]. Both models classify nodes into three classes with availabilities of 0.3, 0.8, and 0.99 but differ in the class size i.e., number of nodes per class.

1) FM1 is based on a one year failure study of a 400-node cluster that observed about 4% nodes accounted for 70% failures, 25% nodes accounted for 20% failures, and 70% nodes accounted for 1% failures [24]. Correspondingly, we approximate class sizes of 5%, 25% and 70%, respectively.
2) FM2 is based on a three month failure study of 240 PlanetLab nodes that showed about 6% nodes had less than 30% availability, 50% nodes had about 80% availability, and 37.7% nodes had more than 99% availability [6]. Correspondingly, we approximate class sizes of 10%, 50%, and 40%, respectively.

We induce cluster failures using these failure probabilities at each application time-step. We expect a typical computing environment to exhibit failure properties between FM1 and FM2 and thus, our results will quantify best and worst-case performance.

B. Data Aggregation Application Results

We implemented an aggregation tree comprising 63 components or processing elements (PEs) [15] on 6 chasses with each PE running on a single host. At each leaf, a source continuously generates data packets and each internal node computes the SUM aggregate of the number of updates in its subtree. We quantify this application's quality metric as the ratio of root's output value to the total data input through all the leaves.

Figure 7 shows the placement for optimal Zen and greedy Zen for this application. Note that both optimal

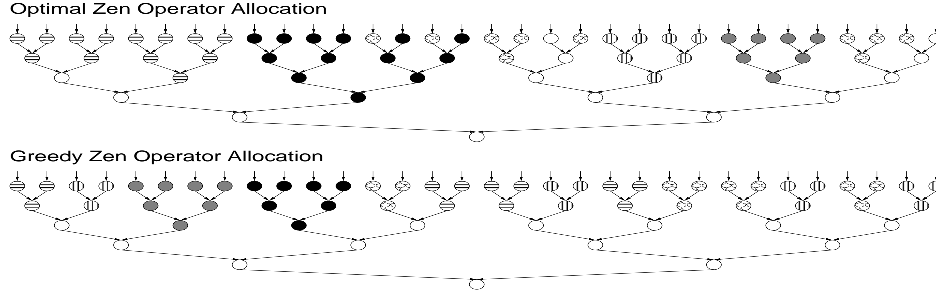

Fig. 7: Optimal Zen and Greedy Zen placement for the data aggregation tree. Each pattern represents one chassis.

and greedy Zen aim to satisfy the consecutive vertex color property—the average number of colors per path is 2 in both cases. The difference between their allocations is because of the most important PEs in the tree (the ones closest to the root.) Greedy Zen places PEs close to the root on one failure unit (chassis); all nodes with depth 0, 1, 2 share the same chassis. In contrast, the optimal algorithm includes paths from two leaves to the root so that the application can produce output even if all the clusters but the one marked white fail.

Figures 8(a) and 8(b) show the effect of optimal Zen, greedy Zen, replication, and optimal Zen+replication on the quality metric by varying the number of failed clusters for FM1 and FM2, respectively. For replication, nodes at levels 0 (root), 1, and 2 are replicated on different machines. Each bar value represents the expected quality, and the following results are computed relative to optimal Zen. For FM1, optimal Zen increases quality by 10%-17% compared to greedy Zen, and 37%-72% compared to replication. For FM2, the corresponding numbers are 7%-11% and 33%-73%, respectively. Since replication only focuses on important components but does not optimize global allocation, it does not yield high benefits under failures that might be a limiting factor for resource-constrained applications. As the number of failed clusters increase, both optimal and greedy Zen show a graceful degradation in the output quality. Finally, Zen+replication outperforms all allocations.

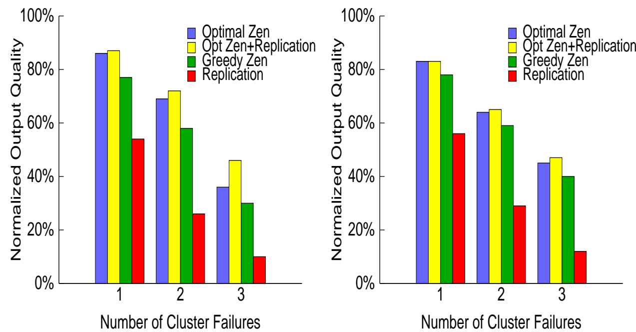

Fig. 8: Comparing optimal Zen, optimal Zen+replication, greedy Zen, and replication for the data aggregation application under failure models FM1(a) and FM2(b).

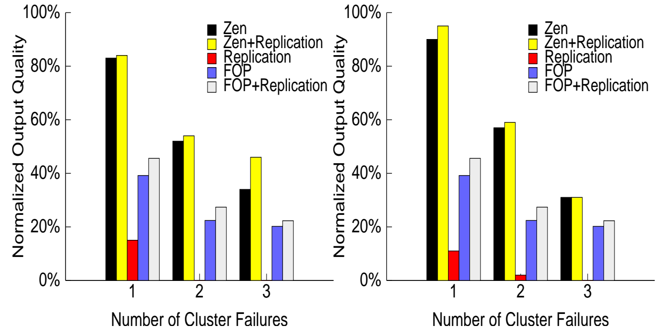

Fig. 10: Normalized Q_{WTTW} for Zen, Zen+replication, replication, FOP, and FOP+replication under FM1(a) and FM2(b).

C. *WTTW Results*

Figure 9 shows the assignments for 39 WTTW PEs using greedy Zen and FOP on 8 chasses. Note that Zen assigns the highest importance PEs, DSN, JAE, PCP, and SGD, on the highest reliability chassis A. Since FOP is failure-oblivious, it assigns them to a less reliable chassis F. The WTTW's quality metric Q_{WTTW} is defined in terms of the accuracy in correctly identifying conversation pairs [28] as:

$$Q_{WTTW} = \frac{(1 - \text{DSN error rate})(\text{No. of pairs reported})}{(\text{No. of conversation pairs})}$$

where DSN is the output PE in Figure 2. We collected results from 600 input conversation streams.

Figures 10(a) and 10(b) show the effect of Zen, Zen+replication, replication, FOP, and FOP+replication on Q_{WTTW} (normalized wrt no-failure quality) for different number of cluster failures under FM1 and FM2, respectively. Each bar value is computed as an average of 8 independent runs, and all numbers below are represented relative to Zen. For FM1, Zen increases quality by 80% compared to replication, 40%-50% compared to FOP, and 35%-45% compared to FOP+replication. For FM2, the corresponding numbers are 90%, 35%-60%, and 30%-50% respectively. The Zen+replication strategy outperforms all other strategies, achieving up to 30% quality increase over Zen.

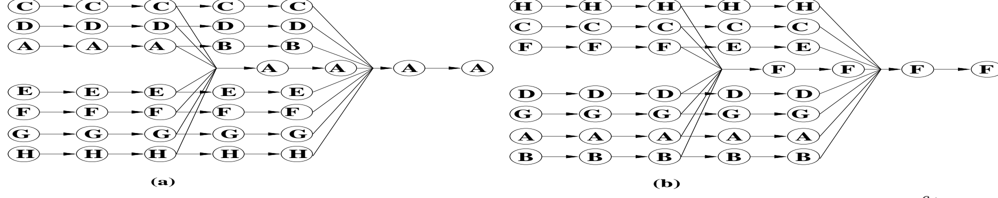

Fig. 9: WTTW PE assignment using Zen (a) and FOP (b) on 8 chasses $(A, \ldots, H)$ ranked (decreasing) by $\frac{c_j}{p_j}(j \in \{A, \ldots, H\})$.

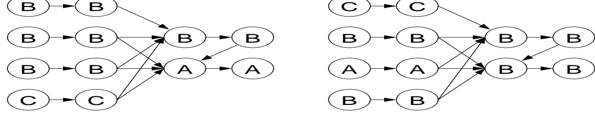

Fig. 11: FAB PE Assignment using Zen (a) and FOP (b) on 3 chasses (A, B, C) ranked (decreasing) by $\frac{c_j}{p_j}(j \in \{A, B, C\})$.

D. FAB results

The third application, FAB, comprises 12 PEs. Compared to data aggregation and WTTW, FAB is (1) small-scale and (2) expected to operate primarily in local area networks that exhibit failure properties closer to the FM1 model. Therefore, we use two FM1-based failure models for FAB: FM1a comprising three failure classes with availabilities 75%, 90%, and 99%, and FM1b with 90%, 95%, and 99%. Our findings, unavoidably, depend on the failure models we used. However, we believe these models are representative for actual deployments of the application.

Figure 11 shows FAB's PE assignments using greedy Zen and FOP on 3 chasses A, B, C ranked (decreasing) by $\frac{c_j}{p_j}(j \in \{A, B, C\})$, and having $2, 8, 4$ nodes, respectively. Note that Zen places the two most important PEs, OTP and RAN, on the most reliable chassis A. FOP does topology-aware but failure-oblivious assignment assigning these PEs to chassis B. For replication, OTP and RAN replicas are placed together on one cluster. The FAB quality metric Q_{FAB} at time t is defined in terms of the accuracy in yield output [27]:

$$Q_{FAB}(t) = \frac{\hat{N}(t)}{N(t)} \left[P_D(t) - 0.2 P_F(t)\right]$$

where $N(t)$ is the total number of wafers, $\hat{N}(t)$ the number of wafers processed by OTP, P_D and P_F are the probabilities of fault-detection and false positives, respectively. We ran FAB on real data from 9000 wafers, and present results averaged across time over 1000 runs.

Figures 12(a) and 12(b) show the effect of all the five component allocations on Q_{FAB} (normalized wrt quality under zero failures) for different number of cluster failures under FM1a and FM2b, respectively. The following results are computed relative to Zen. For FM1a, Zen increases quality by about 43% compared to replication, 17%-20% compared to FOP, and 3%-13% compared to FOP+replication. For FM1b, the corresponding numbers are 37%-45%, 4%-18%, and nearly the same quality as FOP+replication, respectively. As in previous cases, Zen+replication outperforms all strategies, and increases quality by up to 12% over Zen.

In summary, our evaluation shows that for the three real-world PFT applications we considered, Zen's component allocation can significantly reduce the loss in an application output quality under failures.

E. Discussion

Other issues that require further exploration include:
Scalability. Extending the evaluation beyond our current 100 node setup to a large-scale setting. We believe that for larger systems, that preclude manual placement, Zen can provide application and cluster-aware automated component placement.
Topology Constraints. Considering application/external placement constraints (e.g. requirements on placing specific PEs only on particular network hosts). Zen can be used to place unconstrained PEs on the available resources to maximize application quality and failure tolerance, without violating such constraints.
Spatial Failure Correlation. Clustering resources based on failure correlation. Our system model clusters resources that can fail together "spatially". In terms of performance this has been shown to have several benefits [34]. Additional clustering using a combination of spatial and temporal failure correlation (e.g., MTTF, MTBF, MTTR, etc.) may also be used to improve availability.
Performance. Analyzing the interaction between performance ((e.g. *proximity-based*) and availability goals for Zen placement.

VI. Related work

Several studies [6], [10], [24], [25], [34] (and the references therein) have aimed at characterizing the failure properties of real systems. We leverage two important observations common among these studies in our failure model: (1) node failures are not uniformly distributed, and a small fraction of nodes incur most of the failures [6], [24], [34] and (2) node failures are often correlated, with multiple nodes in the system failing nearly simultaneously [10], [21], [24], [25]. Some

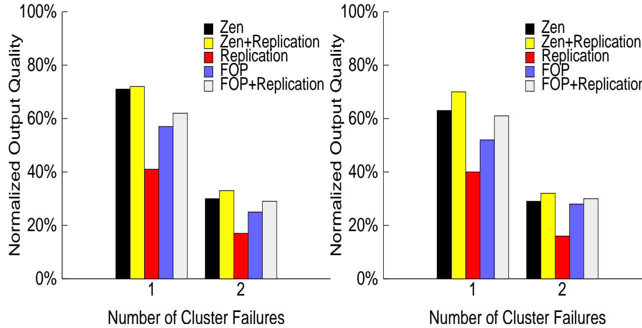

Fig. 12: Normalized Q_{FAB} for Zen, FOP, replication, and Zen+replication under failure models FM1a (a) and FM1b (b).

of these studies have also used knowledge of failure characteristics in resource allocation to improve cluster availability [10], [34].

Similarly, fault-tolerance techniques have been extensively studied for improving availability. E.g., replication, erasure coding, placing replicas on nodes running heterogeneous software versions [17], etc. in distributed storage systems. In task allocation systems, previous solutions [19] have aimed at maximizing the probability of running the *entire* task successfully but not for the PFT model. MapReduce reconfigures the task execution system dynamically under failures by re-executing the failed jobs on available nodes [7]. However, for PFT applications, it is more valuable to keep the application continuously running but allow output of partial results. To our knowledge, there has not been prior work on task allocation for PFT applications.

In stream processing systems, component (operator) placement techniques have generated recent interest primarily for improving performance in resource-limited sensor networks [26] and wide-area stream systems [22], [29]. For achieving high availability, research in this area has focused on replication [4], and storing data and checkpointing [13] but not on failure-aware component allocation. These techniques are complementary to Zen in that it can leverage them to further enhance availability as shown in Section V.

Pietzuch et al. [22] have devised an algorithm to allocate resources in the wide area to optimize network usage of a stream processing system. Cetintemel et al. [29] describe a method to allocate resources in stream processing systems that tries to maximize system usage while not overloading individual machines. While both of these are relevant given that they are addressing issues of resource allocation in stream processing environments, they do not consider the possibility of failure in their allocation strategies.

Finally, two of the design principles in Zen are closely related to the observations by Yu et al. in analyzing availability of multi-object operations [32]: (1) concentrate objects on fewer machines for "strict" operations, and (2) spread objects across machines for "more tolerant" operations. A strict operation can be viewed as a path from a source to a sink requiring availability of *all* in-between nodes, and thus we co-locate them on as few clusters as possible. A more tolerant operation corresponds to multiple parallel paths that should be assigned to different clusters for fault-tolerance.

The area of fault-tolerance for distributed applications is well-researched. The techniques used broadly fall under the categories of checkpoint-restart and replication, and address applications that have a failure model where the failure of one component causes the whole application to fail. We address applications that can continue to run degraded in spite of partial failures and still produce useful output. With such applications, although some critical components may have to be replicated, it is important to appropriately allocate the rest of the components to resources such that the failure of resources causes minimal degradation to the application's output.

Jee et al. [16] describe an algorithm to allocate components to resources for high fault-tolerance in message passing systems. It considers a number of application/system components that are all vying for the same set of resources. There is an inherent assumption here that the number of resources is smaller than the number of components, and that the components have to share the resources in a time-ordered fashion. The algorithm described in this work minimizes the number of components waiting on that resource to be freed. Our model, on the other hand, applies to applications in which all components need to run simultaneously before the application or system can be effective.

The Phoenix recovery system [17] shows that we can group individual "processes" into "clusters" based on which processes tend to fail together, all processes that can fail together being grouped into one cluster. The authors use this model to build a replication strategy for large-scale systems that preserves the system in the face of Internet-wide virus and worm attacks. In particular, they form replica sets consisting of processes picked from many different heterogeneous clusters. While this work uses the dependent-failure model, it mainly concentrates on applications that have a model where the entire application fails if one of its components fails. We focus on a suite of distributed systems that can tolerate partial failures, where an informed placement strategy can help achieve better functionality in spite of failures. Our approach can be used in conjunction with other techniques, such as replication to increase the fault-

tolerance properties of the distributed application.

VII. Conclusions and Future Work

We present Zen, a new failure-aware resource allocation framework to achieve high availability for PFT applications, and show that even simple versions of the component placement problem are NP-Hard. Therefore, Zen provides a 2-approximate polynomial-time algorithm for tree topologies and a greedy algorithm for general graph topologies. Our evaluation shows that for three real-world PFT applications, Zen significantly reduces loss in the application's output quality under failures.

Our future work is to enable self-adaptive component placement to improve availability in large-scale, dynamic environments. Further, we plan to integrate both failure-aware and network-aware (proximity) properties in Zen to achieve optimal component assignment that simultaneously provides high availability and high performance.

References

[1] The PlanetLab distributed testbed. http://www.planet-lab.org.

[2] Streambase. http://www.streambase.com.

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[4] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD*, 2005.

[5] N. Bansal et al. Towards optimal resource allocation in partial-fault tolerant applications (extended), Technical Report, http://www.cs.utexas.edu/users/nav/papers/pft08tr.pdf. 2008.

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

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[8] R. M. Dougall. Availability - What It Means, Why Its Important, and How to Improve It, Sun BluePrints Online. 1999.

[9] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An architecture for a world-wide sensor web. *IEEE Pervasive Computing*, 2(4), 2003.

[10] T. Heath, R. P. Martin, and T. D. Nguyen. Improving cluster availability using workstation validation. *SIGMETRICS*, 2002.

[11] J. Hennessy. The future of systems research. *IEEE Computer*, 1999.

[12] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *VLDB*, 2003.

[13] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *ICDE*, 2005.

[14] J.-H. Hwang, Y. Xing, U. Çetintemel, and S. B. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE*, 2007.

[15] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the Linear Road benchmark on the Stream Processing Core. In *SIGMOD*, 2006.

[16] I. Jee. Optimal fault-tolerant resource allocation in dynamic distributed systems. In *IEEE SPDS*, 1995.

[17] F. Junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. Voelker. Surviving internet catastrophes. In *USENIX*, 2005.

[18] F. Junqueira and K. Marzullo. Coterie availability in sites. In *DISC*, 2005.

[19] S. Kartik and C. S. R. Murthy. Task allocation algorithms for maximizing reliability of distributed computing systems. *IEEE Trans. Comput.*, 46(6):719–724, 1997.

[20] Y. Ling, J. Mi, and X. Lin. A variational calculus approach to optimal checkpoint placement. *IEEE Trans. Comput.*, 2001.

[21] S. Nath, H. Yu, P. Gibbons, and S. Seshan. Subtleties in Tolerating Correlated Failures. In *NSDI*, 2006.

[22] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[23] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. In *Scientific Programming Journal. Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, 2005.

[24] R. Sahoo, M. Squiallnte, A. Sivasubramaniam, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *DSN*, 2004.

[25] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *DSN*, 2006.

[26] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *PODS*, 2005.

[27] D. S. Turaga, O. Verscheure, J. Wong, L. Amini, G. Yocum, E. Begle, and B. Pfeifer. Online fdc control limit tuning with yield prediction using incremental decision tree learning. In *Sematech AEC/APC Symposium XIX*, 2007.

[28] O. Verscheure, M. Vlachos, A. Anagnostopoulos, P. Frossard, E. Bouillet, and P. Yu. Finding 'who is talking to whom' in voip networks via progressive stream clustering. In *ICDM*, 2006.

[29] Y. Xing, J.-H. Hwang, U. Cetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, 2006.

[30] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM*, 2004.

[31] P. Yalagandula, S. Nath, Gibbons, and S. Seshan. Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems. In *WORLDS*, 2004.

[32] H. Yu, P. B. Gibbons, and S. Nath. Availability of multi-object operations. In *NSDI*, 2006.

[33] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *SOSP*, 2001.

[34] Y. Zhang, M. S. Squillante, A. Sivasubramaniam, and R. K. Sahoo. Performance implications of failures in large-scale cluster scheduling. In *JSSPP*, 2004.

APPENDIX

Loss minimization problem is NP-Hardness

We give a proof of Lemma 4.3.

Proof: We give a reduction from the following 3-partition problem. Given a set of $3m$ positive integers $a_1, \ldots, a_{3m}$ and an integer B such that $B/4 < a_i < B/2$ for each $1 \leq i \leq 3m$. The goal is to partition $a_1, \ldots, a_{3m}$ into m sets $S_1, \ldots, S_m$ such that each S_i has sum exactly B. Note that each S_i must have cardinality exactly equal to 3. It is well known that 3-Partition is NP-Hard even when the numbers $a_1, \ldots, a_{3m}$ are polynomially bounded in m, in particular even when $B = O(m^4)$.

Given an instance of 3-Partition, we define an instance of our problem as follows. We have $m+1$ colors (nodes) each with capacity B. There are $3m$ paths $P_1, \ldots, P_{3m}$ of lengths $a_1, \ldots, a_{3m}$ respectively. One end of each of the paths is joined to a vertex r to yield a tree rooted at r. For each of these paths, v_l, the value entering the leaf is 1. Additionally, we have a special path P_0 of length $B-1$ with one end connected to r with $v_l = 6m$.

We claim that there is a solution with value $12m$ if and only if the 3-partition problem has a solution. Given a solution $S_1, \ldots, S_m$ to the 3-partition problem, for $i = 1, \ldots, 3m$ we color P_i with color j if $a_i \in S_j$. We color the path $P_0 \cup \{r\}$ with color 0. Since, $d(l) = 1$ for the path $P_0 \cup \{r\}$ and 2 for the paths $P_i \cup \{r\}$, for $1 \leq i \leq 3m$, it follows that the loss is equal to $12m$.

To show the converse, we first observe that the path $P_0 \cup \{r\}$ must be monochromatic otherwise this path has cost at least $12m$ and the overall solution has cost at least $15m$ (since each path $P_i \cup \{r\}$ for $1 \leq i \leq 3m$ will also contribute at least 1). Without loss of generality, suppose that $P_0 \cup \{r\}$ has color 0. As $|P_0 \cup \{r\}| = B$ and each node has capacity B, no other node can be colored 0. Thus, every path P_i must use a color other than 0, and hence the path $P_i \cup \{r\}$ must have $d(l) \geq 2$. Thus, the solution has value $12m$ only if each path P_i is mono-chromatic which implies an exact 3-partition. ■

Figure for proof of consecutive vertex color property

In Figure 13(a), components 1 and 6 are colored gray, while component 3 is white. However, if we swap the colors of 3 and 6, as in Figure 13(b), the number of colors on the path 6-3-1 stays the same, i.e. 2, but the number of colors on path 7-3-1 *goes down* by 1.

Dynamic programming Formulation

We describe here the details of the dynamic program used to compute an optimum solution when unbounded resources are available.

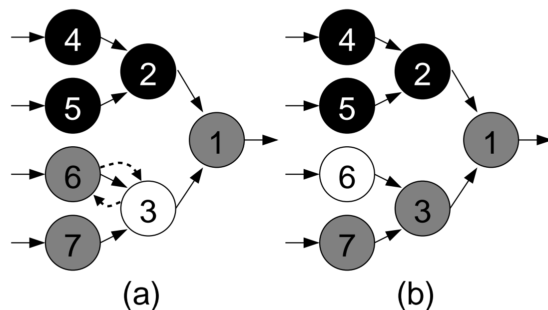

Fig. 13: The consecutive vertex color property.

Given the tree T, for each vertex w in T, let T_w denote the subtree rooted at w. Consider the following dynamic programming formulation. For every vertex w, and integer $q = 1, \ldots, c$, let $V(w, q)$ denote the loss-minimizing coloring of T_w subject to the constraint that the color used for vertex w is used at most q times in this tree. In this notation, our goal is to compute $V(r, c)$.

A naive (possible exponential time) way to perform the dynamic program for the loss minimizing problem is the following: Suppose our goal is to compute $V(w, q)$. If w is a leaf we simply set $V(w, q) = v_w$. Hence, suppose w is an internal node and let $w_1, \ldots, w_k$ be the children of w. Consider all possible ordered partitions of $q-1$ into k parts. That is, consider $q_1, \ldots, q_k$ such that $q_i \geq 0$ for $1 \leq i \leq k$ and $\sum_i q_i = q - 1$. Given a fixed partition $X = (q_1, \ldots, q_k)$, let $J(X)$ denote the set of indices j for which $q_j = 0$. Consider the partition X among all possible partitions that minimizes

$$\sum_{i \notin J(X)} V(w_i, q_i) + \sum_{i \in J(X)} (V(w_i, c) + \sum_{l \in L_i} v_l)$$

Note that since each cluster has capacity c, $V(w_i, c)$ is just the optimum achievable loss for T_{w_i}. By the above discussion, this is a valid dynamic program as the values $V(w, q)$ can be computed starting from the leaves in a bottom up fashion. However, since there could be as many as $\Omega(q^k)$ partitions of $q-1$ to consider, the running time per step of the dynamic program would be exponentially large.

To get around the problem of needing to compute an exponential number of partitions, we use the following trick: We define the quantity $\tilde{V}(w, q', j)$ which indicates the total optimum possible loss at w due to the subtrees $T_{w_1}, T_{w_2}, \ldots, T_{w_j}$ subject to the constraint that the color for w is used at most q' times in $T_{w_1}, \ldots, T_{w_j}$. In this notation, computing $V(w, q)$ is equivalent to computing $\tilde{V}(w, q-1, k)$, where k is the degree of w. To compute $\tilde{V}(w, q', j)$, suppose we have already computed $\tilde{V}(w, q'', j-1)$ for all $q'' = 1, \ldots, q'-1$ and also precomputed $\tilde{V}(w_i, q'', j_i)$ (and hence $V w_i, q''$) for all

children w_i and $j_i = 1, \ldots, k_i$ where k_i is the degree of w_i . Now $V(w, q', j)$ is obtained by assigned either zero copies of the color of w to T_{w_j} in which case

$$\tilde{V}(w, q', j) = \tilde{V}(w, q', j-1) + V(w_i, c) + \sum_{l \in L_i} v_l$$

or else some $h > 0$ copies of the color for w are used in T_j in which case

$$\tilde{V}(w, q', j) = \min_{h=1}^{q'} (V(T_j, h) + \tilde{V}(T', q' - h, j-1))$$

Thus $\tilde{V}(w, q', j)$ is simply the minimum of these two quantities. This gives a valid dynamic programming formulation for V.

We now analyze the running time and space requirements of this dynamic program. First, the quantity $\sum_{l \in L_i} v_l$ can be precomputed for each subtree T_w, by a simple bottom up algorithm in $O(n)$ time. In the dynamic program, for each vertex in the tree store we compute the quantities $V(w, q', j)$ for all values of j up to the degree of the vertex and q' up to the capacity of the cluster. Since the sum over all the degrees in the tree is $O(n)$, the overall (amortized) space requirement is $O(n^2 c)$. Moreover, the algorithm requires $O(q') = O(c)$ time to compute $V(w, q', j)$ from the previous entries $V(w, q'', j-1)$. Thus the running time is $O(c)$ times the space requirement and hence $O(n^2 c^2)$.