Improving Availability in Large, Distributed Component-Based Systems via Redeployment

Marija Mikic-Rakic[2], Sam Malek[1,3], and Nenad Medvidovic[1]

[1]University of Southern California, Computer Science Department, Los Angeles, CA, 90089, USA. {malek, neno}@usc.edu

[2]Google Inc., Santa Monica, CA, 90405, USA. marija@google.com

[3]The Boeing Company, 5301 Bolsa Avenue, Huntington Beach, CA, 92647, USA. sam.malek2@boeing.com

Abstract. In distributed and mobile environments, the connections among the hosts on which a software system is running are often unstable. As a result of connectivity losses, the overall availability of the system decreases. The distribution of software components onto hardware nodes (i.e., the system's deployment architecture) may be ill-suited for the given target hardware environment and may need to be altered to improve the software system's availability. The critical difficulty in achieving this task lies in the fact that determining a software system's deployment that will maximize its availability is an exponentially complex problem. In this paper, we present a fast approximative solution for this problem, and assess its performance. In addition to significantly improving availability, our solution, in general, also reduces the overall interaction latency in the system. We evaluate our solution on a large number of automatically generated application scenarios.

1 Introduction

The emergence of mobile devices, such as portable computers, PDAs, and mobile phones, and the advent of the Internet and various wireless networking solutions make computation possible anywhere. Applications involving these mobile devices are highly dependent on the underlying network. Unfortunately, network connectivity failures are not rare: mobile devices face frequent and unpredictable connectivity losses due to their constant location change and lack of network coverage; the costs of wireless connectivity often also induce user-initiated disconnection; and even the highly reliable WAN and LAN connectivity is unavailable 1.5% to 3.3% of the time [24].

For this reason, highly distributed and mobile systems are challenged by the problem of *disconnected operation* [22], where the system must continue functioning in the temporary absence of the network. This presents a major challenge for software systems that are highly dependent on network connectivity because each local subsystem is usually dependent on the availability of non-local resources. Lack of access to a remote resource can make a particular subsystem, or even the entire system unusable.

A software system's *availability* is commonly defined as the degree to which a system is operational and accessible when required for use [7]. In the context of highly distributed, mobile environments, where the most common cause of (partial) system inaccessibility is network failure [23], we quantify availability as the ratio of the number of successfully completed inter-component interactions in the system to the total number of attempted interactions over a period of time.

In this context, a key observation is that the distribution of software components onto hardware nodes (i.e., a system's software *deployment architecture*, illustrated in Figure 1.) greatly influences the system's availability in the face of connectivity losses. For example, in such cases it is desirable to collocate components that interact frequently. However, the parameters that influence the optimal distribution of a system (e.g., the reliability of network links) may not be known before the system's deployment. For this reason, the (initial) software deployment architecture may be ill-suited for the given target hardware environment. This means that a *redeployment* of the software system may be necessary to improve its availability.

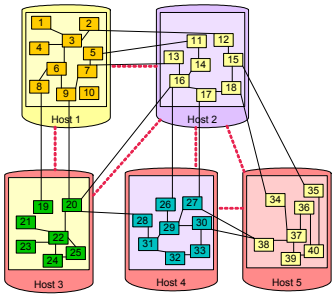


Figure 1. A sample deployment architecture with five hardware hosts and 40 software components. Dotted lines represent network connectivity, while solid lines represent interacting components.

There are several existing techniques that can support various subtasks of redeployment, such as monitoring [4] to assess hardware and software properties of interest, component migration [3] to facilitate redeployment, and dynamic system manipulation [21] to effect the redeployment once the components are migrated to the appropriate hosts. However, the critical difficulty in achieving this task lies in the fact that *determining* a software system's deployment that will maximize its availability (i.e., the *optimal* deployment) is an exponentially complex problem: in the most general case the complexity is k^n, where k is the number of hardware hosts and n the number of software components. Existing approaches that recognize this (e.g., I5 [1]) still assume that all system parameters are known beforehand and that infinite time is available to calculate the optimal deployment. Other approaches, such as Coign [6], restrict their solution to two hosts and client-server architectures, thus decreasing the algorithm's complexity, but also the resulting solution's usefulness.

For most practical cases finding the optimal deployment is infeasible: it requires an exponentially-complex "exact" algorithm. This paper presents an approximative algorithm, *Avala*, for increasing a system's availability by estimating the system's redeployment architecture in polynomial time. We provide a detailed assessment of Avala's performance. Since for large systems the optimal redeployment cannot be calculated in a reasonable amount of time, we compare the availability achieved via our solution to the availability of a system's "most likely" deployment. We present two additional algorithms that we have developed to obtain the availability of the most likely deployment. Finally, in addition to significantly improving the overall system availability, we show that Avala, in general, also reduces the overall interaction latency in the system.

The Avala algorithm is part of an integrated solution to increasing a system's availability [13,17,12], which enables the three key redeployment tasks: (1) monitoring the system to gather the data that influences its availability; (2) estimating the redeployment architecture; and (3) effecting that architecture.

The remainder of the paper is organized as follows. Section 2 defines the problem our work is addressing and discusses a set of assumptions in our approach. Section 3 presents an overview of the related work and of our overall redeployment approach. Section 4 describes the exact algorithm and discusses its complexity. Section 5

describes the Avala algorithm for the exponentially complex redeployment problem. Section 6 presents our approach for evaluating Avala, the results of its assessment, and our tool support. Section 7 discusses the characteristics as well as current limitations of the Avala algorithm. The paper concludes with a discussion of future work.

2 The Redeployment Problem

The distribution of software components onto hardware nodes (i.e., a system's software *deployment architecture*) greatly influences the system's availability in the face of connectivity losses. For example, components located on the same host will be able to communicate regardless of the network's status, which is not the case with components distributed across different hosts. However, the reliability of connectivity (i.e., the rate of failure) among the "target" hardware nodes on which the system is deployed is usually not known before the deployment. The frequencies of interaction among software components may also be unknown. Hence, the initial software deployment architecture may be ill-suited for the given target hardware environment. This means that a *redeployment* of the software system may be necessary to improve its availability.

The critical difficulty in achieving this task lies in the fact that determining a software system's deployment architecture that will maximize its availability (referred to as *optimal deployment architecture*) is an exponentially complex problem.

2.1 Problem Definition

In addition to the characteristics of hardware connectivity and software interaction, there are other constraints on a system's redeployment, including: (1) the available memory on each host; (2) the required memory for each software component; and (3) possible restrictions on component locations (e.g., two CPU-intensive components may not be allowed to reside on the same host).

Figure 2. shows a formal model that captures the system properties and constraints, and a formal definition of the problem. The mem_{comp} function captures the required memory for each component. The frequency of interaction between any pair of components is captured via the *freq* relation. Each host's available memory is captured via the mem_{host} function. The reliability of the link between any pair of hosts is captured via the *rel* relation. Using the *loc* relation, deployment of any component can be restricted to a subset of hosts, thus denoting a set of *allowed* hosts for that component. Using the *colloc* relation constraints on collocation of components can be specified.

The definition of the problem contains the criterion function A, which formally describes a system's availability as the ratio of the number of successfully completed interactions in the system to the total number of attempted interactions. Function f represents the exponential number of the system's candidate deployments. To be considered valid, each candidate deployment must satisfy the three conditions. The first condition in the definition states that the sum of memories of the components that are deployed onto a given host may not exceed the available memory on that host. The second condition states that a component may only be deployed onto a host that belongs to a set of allowed hosts for that component, specified via the *loc* relation. Finally, the third condition states that two components must be deployed onto the same host (or on different hosts) if required by the *colloc* relation.

2.2 Assumptions

The problem defined in Section 2.1 is an instance of the more general redeployment problem, described in [16]. In this paper, we consider a subset of all possible constraints, and a specific criterion function, which is to maximize the system's availability. Through the *loc* and *colloc* functions, one can include other constraints (e.g., security, CPU, bandwidth), not directly captured in our problem description. However, if multiple resources, such as bandwidth and CPU, are as restrictive as memory in a given system, then capturing them only via the *loc* and *colloc* functions will not be sufficient. In [16] we describe how such cases could be addressed, by introducing additional system parameters into the model and introducing additional constraints that a valid deployment should satisfy. For example, for systems where the network bandwidth and volume of exchanged data severely restrict the number of possible deployments, the formal problem statement would need to include two additional constraint relations and an additional condition: (1) relation *evt_size* to capture the average size of data exchanged between a pair of components; (2) relation *bw* to capture the bandwidth

Model

Given:

(1) a set C of *n* components ($n = |C|$), a relation $freq : C \times C \rightarrow \Re$, and a function $mem_{comp} : C \rightarrow \Re$

$$freq(c_i, c_j) = \left\{ \begin{array}{l} 0 \quad if \ c_i = c_j \\ frequency\ of\ communication\ between\ c_i\ and\ c_j \quad if \quad c_i \neq c_j \end{array} \right\}$$

$$mem_{comp}(c) = required\ memory\ for\ c$$

(2) a set H of *k* hardware nodes ($k = |H|$), a relation $rel : H \times H \rightarrow \Re$, and a function $mem_{host} : H \rightarrow \Re$

$$rel(h_i, h_j) = \left\{ \begin{array}{l} 1 \quad if \quad h_i = h_j \\ 0 \quad if \quad h_i \ is\ not\ connected\ to\ h_j \\ reliability\ of\ the\ link\ between\ h_i\ and\ h_j \quad if \quad h_i \neq h_j \end{array} \right\}$$

$$mem_{host}(h) = available\ memory\ on\ host\ h$$

(3) Two relations that restrict locations of software components $loc : C \times H \rightarrow \{0,1\}$ $colloc : C \times C \rightarrow \{-1,0,1\}$

$$loc(c_i, h_j) = \left\{ \begin{array}{l} 1 \quad if \quad c_i\ can\ be\ deployed\ onto\ h_j \\ 0 \quad if \quad c_i\ cannot\ be\ deployed\ onto\ h_j \end{array} \right\}$$

$$colloc(c_i, c_j) = \left\{ \begin{array}{l} -1 \quad if \quad c_i\ cannot\ be\ on\ the\ same\ host\ as\ c_j \\ 1 \quad if \quad c_i\ has\ to\ be\ on\ the\ same\ host\ as\ c_j \\ 0 \quad if \quad there\ are\ no\ restrictions\ on\ collocation\ of\ c_i\ and\ c_j \end{array} \right\}$$

Definition

Problem:

Find a function $f : C \rightarrow H$ such that the system's overall availability A defined as

$$A = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} \left(freq(c_i, c_j) * rel(f(c_i), f(c_j)) \right)}{\sum_{i=1}^{n} \sum_{j=1}^{n} freq(c_i, c_j)}$$

is maximized, and the following three conditions are satisfied:

(1) $$\forall i \in [1,k] \left\{ \forall j \in [1,n] \quad f(c_j) = h_i \quad \middle| \quad \sum_j mem_{comp}(c_j)) \leq mem_{host}(h_i) \right\}$$

(2) $$\forall j \in [1,n] \qquad loc(c_j, f(c_j)) = 1$$

(3) $$\forall k \in [1,n] \quad \forall l \in [1,n] \qquad (colloc(c_k, c_l) = 1) \Rightarrow (f(c_k) = f(c_l))$$
$$(colloc(c_k, c_l) = -1) \Rightarrow (f(c_k) \neq f(c_l))$$

In the most general case, the number of possible functions *f* is k^n. However, note that some of these deployments may not satisfy one or more of the above three conditions.

Figure 2. Formal statement of the problem.

between a pair of hosts; and (3) the following condition:

$$(\forall i \in [1,k] \quad \forall j \in [i+1,k])$$
$$\left(\begin{array}{l} (\forall l \in [1,n] \quad \forall m \in [l+1,n]) \\ \left(\begin{array}{l} where\ f(c_l) = h_i \wedge f(c_m) = h_j \\ \sum_{l,m} data\_vol(c_l, c_m) \le effective\_bw(h_i, h_j) \end{array} \right) \end{array} \right) \begin{array}{ll} where & data\_vol(c_x, c_y) = freq(c_x, c_y) * evt\_size(c_x, c_y) \\ and & effective\_bw(h_x, h_y) = rel(h_x, h_y) * bw(h_x, h_y) \end{array}$$

This condition states that, for each network link between a pair of hosts, the total volume of data exchanged across that link does not exceed the link's effective bandwidth. The algorithm presented in this paper would need to be altered to ensure the satisfaction of this condition.

Our definition of availability considers all inter-component interactions equally important. For systems in which this may not be the case, the same model and algorithm can still be used: the *freq* relation can be changed to correspond to the product of interaction frequency and importance of data, and the remainder of the model and problem definition would remain unchanged.

The problem presented in Section 2.1 is also based on the assumption that system parameters are stable over a given period of time *T*, during which we want to improve the system's availability.[1] It also relies on the assumption that the time required to perform the system's redeployment is negligible with respect to *T*. Otherwise, the system's parameters would be changing too frequently and the system would undergo continuous redeployments to improve the availability for parameters that change either before or shortly after the redeployment is completed.We believe this to be a reasonable assumption, which is reflective of a number of existing systems (e.g., see [20]).

Finally, our approach relies on the assumption that the given system's deployment architecture is accessible from some central location. We realize that this assumption may not be justified in a class of software systems that are decentralized, and have developed a decentralized solution that is complementary to this work [11]. However, in a centralized system, the algorithm can leverage the availability of global knowledge about system parameters on a central host to run more efficiently than a decentralized algorithm (in terms of required computational and communicational resources). Therefore, when dealing with a centralized system, it is preferable to use a centralized solution instead of a more generally applicable decentralized solution.

3 Background and Related Work

In this section we present a brief overview of disconnected operation approaches, and provide an in-depth look at three approaches that have specifically focused on the system redeployment problem. Additionally, to provide the context for Avala, we present an overview of our overall approach.

3.1 Disconnected Operation

We have performed an extensive survey of existing disconnected operation approaches, and provided a framework for their classification and comparison [18]. The most commonly used techniques for supporting disconnected operation are caching [9], hoarding [10], queueing remote interactions [6], and multi-modal components

1. We do not require that system parameters be constant during *T*, but assume that each parameter can be approximated with its average over *T*, with an error no greater than a given threshold ε [14,17].

[22]. None of these techniques changes the system's deployment architecture. Instead, they strive to improve the system's availability by sacrificing either correctness (in the case of replication) or service delivery time (queueing), or by requiring implementation-level changes to the existing application's code [22].

3.2 Redeployment

I5 [1] proposes the use of the binary integer programming model for generating an optimal deployment of a software application over a given network. I5 is applicable only to systems with very small numbers of software components and target hosts, and to systems whose characteristics, such as frequencies of component interactions, are known at design time and are stable throughout the system's execution.

Coign [6] provides a framework for distributed partitioning of COM applications across the network. Coign employs the lift-to-front minimum-cut graph cutting algorithm to choose a deployment architecture that will result in minimal overall communication time. However, Coign can only handle situations with two-host, client-server applications. Coign recognizes that the problem of distributing an application across three or more hosts is NP hard and does not provide solutions for such cases.

Kichkaylo et al. [8], provide a model, called component placement problem (CPP), for describing a distributed system in terms of network and application properties and constraints, and an AI planning algorithm, called Sekitei, for solving the CPP model. CPP does not provide facilities for specifying the goal, i.e., a criterion function that should be maximized or minimized. Therefore, Sekitei only searches for any valid deployment that satisfies the specified constraints, without considering the quality of a found deployment.

3.3 Our Overall Approach

The Avala algorithm described in this paper is part of an integrated solution for increasing the availability of a distributed system during disconnection [14,15,17,12], without the shortcomings of the existing approaches. For instance, unlike [22] our approach does not require any recoding of the system's existing functionality or human intervention; unlike [9] it does not sacrifice the correctness of computations; in comparison to [6] it minimizes service delivery delays; finally, unlike any of the existing redeployment approaches, our approach scales to very large systems with arbitrary topologies. We directly leverage a software system's *architecture* in accomplishing this task. We support runtime redeployment to increase the software system's availability by (1) monitoring the system, (2) estimating its redeployment architecture, and (3) effecting the estimated redeployment architecture. We provide lightweight facilities for runtime monitoring [17,12] to extract the system's model (recall Figure 2.). The monitoring information is then used by Avala to estimate the improved deployment architecture. Finally, we provide a set of automated deployment facilities [15,12] to effect the estimated architecture.

4 Exact Algorithm

One can ensure that she will find a system's optimal deployment by trying all possible deployments of components onto hosts. The selected deployment is the one that has the maximum availability (referred to as *exact maximum*) and that satisfies the constraints posed by memory and restrictions on the locations of software components. This "exact" algorithm guarantees at least one optimal deployment. The complexity of

this algorithm in the general case (i.e., with no restrictions on component locations) is $O(k^n)$, where k is the number of hardware hosts, and n the number of software components. By fixing a subset of m components to selected hosts, the complexity of the exact algorithm reduces to $O(k^{n-m})$. Even with this reduction, this algorithm may be computationally too expensive unless the number of hardware nodes and unfixed software components is very small. For example, even for a relatively small deployment architecture (15 components, 4 hosts), a Java JDK 1.4 implementation of the exact algorithm runs for more than eight hours on a mid-range PC.

5 The Avala Algorithm

Given the complexity of the exact algorithm, we had to devise an approximative algorithm that would significantly reduce this complexity while exhibiting good performance. In this section, we describe and assess the performance of Avala, an approximative algorithm with polynomial time complexity. Avala leverages a greedy approach [2].

Pseudo-code of Avala is provided in Figure 3. Avala incrementally assigns software components to the hardware hosts. At each step of the algorithm, the goal is to select the assignment that will maximally contribute to the availability function. This is achieved by selecting the "best" host and "best" software component at each step.

Avala starts by ranking all hardware nodes and software components. The initial ranking of hardware nodes is done by calculating *initHostRank* for each hardware node i, as follows:

$$initHostRank_i = a * \sum_{j=1}^{k} rel(h_i, h_j) + b * mem_{host}(h_i)$$

where a and b are calibration factors that denote the respective contributions of link reliability and memory to the selection of the "best" host. In Section 6 we discuss how varying a and b influences the algorithm's performance.

The ranking of software components is done by calculating *initCompRank* for each component i, as follows:

$$initCompRank_i = d * \sum_{j=1}^{n} freq(c_i, c_j) + \frac{e}{mem_{comp}(c_i)}$$

where d and e denote the respective contributions of event frequency and memory to the selection of the "best" component. In Section 6 we also discuss how varying d and e influences the algorithm's performance.

After the initial ranking is performed, the host with the highest value of *initHostRank* is selected as the current host h. A component with the highest value of *initCompRank* that satisfies the *mem* and *loc* constraints (conditions 1 and 2 in Figure 2.) is selected and assigned to h. The next software component(s) to be assigned to h are the ones with smallest required memory whose placement on h would maximally contribute to the availability function, i.e., the components with the highest volumes of interaction with the component(s) already assigned (mapped) to h. The selection is performed by calculating the value of *compRank* for each unassigned component as follows:

$$compRank(c_i, h) = d * \sum_{j=1}^{numOfMappedComps} \left(freq(c_i, mC(j)) * rel(h, f(mC(j))) \right) + \frac{e}{mem_{comp}(c_i)}$$

where *mC(j)* is shorthand for *mappedComponents(j)*. The selected component is the one with the highest value of *compRank* that satisfies memory, *loc*, and *colloc* constraints with respect to the current host h and components already assigned. This process repeats until h is full (i.e., there is no component small enough to fit on h).

$$hostRank(h_i) = a * \sum_{j=1}^{numOfMappedHosts} rel(h_i, mappedHost(j)) + b * mem_{host}(h_i)$$

The next host to be selected is the one with the highest memory capacity and highest link quality (i.e., highest value of *hostRank*) with the host(s) already selected:

The process of selecting software components repeats, until all the components have been assigned to a host.

The complexity of the Avala algorithm in the most general case (i.e., when the number of components fixed to a single host is zero, and there are more components than hosts) is $O(n^3)$, derived as follows:

O(*Avala_algorithm*) = O((n-1) * (O(*next_comp*) + O(*next_host*))) =
O((n-1) * (n * O(*compRank*) + k * O(*hostRank*)) = O((n-1) * (n * n + k * k)) =
$O(n^3 + n * k^2) = O(n^3)$, if n>k

Note that if there are few or no constraints on component location, and total available memory on hosts is significantly above the total required memory by the components, some of the hosts will get filled to their capacity, while others may contain few components or even be empty. The uneven distribution of components among hosts results in higher overall availability of the system since it utilizes the maximum reliability for interactions between components residing on the same host. However, it may also result in undesirable effects on the system, such as overloading the CPUs on hosts with large numbers of components, or overloading the used subset of network links. The Avala algorithm currently addresses this concern only via the *loc* and *colloc* constraints (e.g., by assigning a UI component to each host). However, as described in Section 2.2, both the problem statement and the algorithm could be modified to take other criteria (e.g., CPU, bandwidth) into consideration.

The contributions of Avala are two fold. By separating the component and host selection process from the remaining algorithm's logic, we can easily extend the algorithm to include other system parameters and constraints. Secondly, by parameterizing the selection process for components and hosts along two separate dimensions (memory and frequency in the case of components, and memory and reliability in the case of hosts) the algorithm can automatically adapt to variations in input parameters.

6 Evaluation

Due to the exponential nature of the deployment problem, evaluating Avala's results against the exact solution is only feasible for very small systems (e.g., less than 15 components and 4 hosts). In these cases, the exact algorithm can also produce the average availability of all the deployments (referred to as *exact average*), thus providing an additional criterion for evaluation. However, we still need to assess how well the Avala algorithm performs for systems with (much) larger numbers of components and hosts. To that end, we use two additional algorithms discussed below.

6.1 Evaluation Criteria

We have developed a stochastic algorithm (called *unbiased* stochastic algorithm) that randomly selects a subset of all possible deployments, and uses the availabilities of these deployments to estimate the average availability of a given system. The obtained average availability corresponds to the system's "most likely" availability. The unbiased stochastic algorithm generates different deployments by randomly assigning each component to a single host from the set of available hosts for that component. If the

randomly generated deployment satisfies all the constraints, the availability of the produced deployment architecture is calculated. This process repeats a given number of times, and the average availability (referred to as *unbiased average*) and maximum availability (referred to as *unbiased maximum*) are calculated. The complexity of calculating the availability for each valid deployment is quadratic (recall Figure 2.), resulting in the same complexity of the overall unbiased stochastic algorithm ($O(n^2)$).

In addition to this algorithm, for the sake of completeness we also compare Avala's results against another stochastic algorithm (called *biased* stochastic algorithm) that we have developed and assessed previously [14]. The biased stochastic algorithm randomly orders the hosts and randomly orders the components. Then, going in order, it assigns as many components to a given host as can fit on that host (due to memory constraints), also ensuring that the *loc* and *colloc* constraints are satisfied. Once the host is full, the algorithm proceeds with the same process for the next host in the ordered list of hosts, and the remaining unassigned components in the ordered list of components, until all components have been deployed. This process repeats a given number of times, and the average availability (referred to as *biased average*) and maximum availability (referred to as *biased maximum*) are calculated. The complexity of this algorithm is also polynomial, since we need to calculate the availability for every deployment, and that takes $O(n^2)$ time.

```
next _host (unmappedHosts)                                          O(k^2)
    bestHostRank ← 0
    bestHostIndex ← -1
    for idx to unmappedHosts.length
      thisHostRank ← hostRank(unmappedHosts[idx])
        if bestHostRank < thisHostRank
          bestHostIndex ← idx
          bestHostRank ← thisHostRank
  if bestHostIndex=-1 return NULL
  else return unmappedHosts[bestHostIndex]
next_comp(comps, unmappedComps, currentHost)                        O(n^2)
  bestCompRank ← 0
  bestCompIndex ← -1
  mappedComps ← comps – unmappedComps
  for idx to unmappedComps.length
    if (unmappedComps[idx].memory <= currentHost.memory
    and unmappedComps[idx] sastisfies loc
        and colloc constraints with mappedComps)
      thisCompRank ← compRank(unmappedComps[idx],
                      currentHost)
      if bestCompRank < thisCompRank
        bestCompIndex ← idx
        bestCompRank ← thisCompRank
  if bestCompIndex=-1 return NULL
  else return unmappedComps[bestCompIndex]
avala_algorithm (hosts, comps)                                      O(n^3)
  numOfHosts ← hosts.length
  numOfComps ← comps.length
  numOfMappedComps ← 0
  unmappedComps ← comps
  h ← host with max(initHostRank)
  unmappedHosts ← hosts – h
  numOfMappedHosts ← 1
  c ← component with max(initCompRank where loc(c,h)=1)
  while (numOfMappedComps < numOfComps
  and numOfMappedHosts < numOfHosts and h<>-1)
    while (h.memory>c.memory
    and numOfMappedComps < numOfComps and c<>-1)
      unmappedComps ← unmappedComps – c
      numOfMappedComps ← numOfMappedComps + 1
      h.memory ← h.memory – c.memory
      deployment ← deployment U (deploy c to h)
      c ← next_comp(comps,unmappedComps,h)
    h ← next_host(unmappedHosts)
    unmappedHosts ← hosts – h
    numOfMappedHosts ← numOfMappedHosts + 1
if numOfMappedComps= numOfComps return deployment
else NO DEPLOYMENT WAS FOUND
```

Figure 3. Pseudo-code of the Avala algorithm (left) and its complexity (right).

6.2 Testing Environment

To assess Avala's performance, we have leveraged DeSi [15], a visual deployment

environment that supports specification, manipulation, visualization, and (re)estimation of deployment architectures for large-scale, highly distributed systems. DeSi provides users with a graphical front-end to input values for numbers of hosts and software components as well as the ranges for available memory on the hosts, required memory for the components, frequency of interaction between components, and reliability of connectivity between hosts. DeSi uses this information to randomly generate a *redeployment problem* by fixing all hardware and software parameters needed as inputs to the algorithms. DeSi provides the ability to invoke different redeployment algorithms and display their results. Finally, the algorithms can be benchmarked a given number of times: DeSi iteratively generates different redeployment problems a specified number of times using the same set of ranges for input data, invokes each one of the algorithms for each problem, and calculates the average results.

DeSi provides a number of additional facilities for visualizing and graphically manipulating a system's deployment architecture, as well as several host- and component-specific views. A discussion of these facilities is outside the scope of this paper, however, and can be found in [15].

6.3 Evaluation Results

We have assessed the performance of the Avala algorithm by comparing it against the exact algorithm and the two stochastic algorithms for systems with small numbers of components and hosts (i.e., less than 13 components, and less than 5 hosts).

In large numbers of randomly generated redeployment problems, the Avala algorithm invariably found a solution that was at least 90% of the optimal (i.e., the *exact maximum*). In Table 1, we present results of 5 different redeployment problems, as well as the average results for 30 different randomly generated problems (using the DeSi's benchmark option and shown in the right-most column).[1] The average improvement of availability by Avala over the *exact average* was 34.7%.

Table 1: Comparing the performance of Avala for different architectures with 10 components and 4 hosts.

	10 comps 4 hosts 1 iteration	10 comps 4 hosts 1 iteration	10 comps 4 hosts 1 iteration	10 comps 4 hosts 1 iteration	10 comps 4 hosts 1 iteration	10 comps 4 hosts 30 iterations
Unbiased maximum	0.790	0.732	0.636	0.763	0.932	0.742
Unbiased average	0.560	0.558	0.605	0.516	0.581	0.585
Biased maximum	0.621	0.701	0.615	0.679	0.745	0.738
Biased average	0.572	0.551	0.606	0.544	0.633	0.626
Exact maximum	0.895	0.800	0.733	0.985	0.983	0.820
Exact average	0.558	0.555	0.628	0.513	0.580	0.585
Avala	***0.854***	***0.792***	***0.673***	***0.984***	***0.962***	***0.788***
% improvement over the exact average[a]	53.0	42.7	7.2	91.8	65.9	34.7
% improvement over the unbiased average[b]	52.5	41.9	11.2	90.6	65.5	34.7
% of the exact[c]	95.4	99	91.8	99.9	97.9	96.1

a. calculated as 100% * (Avala – exact average) / exact average
b. calculated as 100% * (Avala – unbiased average) / unbiased average
c. calculated as 100% * Avala / exact maximum

For larger problems, where the exact algorithm is infeasible, we have compared the results of Avala against the results of the stochastic algorithms. In [19], we demonstrated that increasing the number of iterations beyond 10,000 does not significantly change the average availability of the two stochastic algorithms. Thus, the stochastic algorithms were executed with 10,000 iterations for larger deployment

1. The highlighted columns in Tables 1 and 2 will be discussed further in Section 7.

problems.

Table 2 illustrates the results of 6 different benchmarks where the number of components was varied between 30 and 1000 and number of hosts between 7 and 100. The average relative improvement of availability produced by Avala was 33.9% over the *unbiased average*, 30% over the *unbiased maximum*, 28% over the *biased average*, and 11% over the *biased maximum*. Avala also produced its results quickly. For illustration, it took 38 seconds to solve the largest problem (100 hosts and 1000 components) on a mid-range PC; by comparison, the exact algorithm would require over 10^{1984} *years* to determine the optimal deployment. Solving the same problem on a high-end computer (2.8GHz Pentium 4) reduces Avala's running time over 10-fold.

Table 2: Comparing the performance of the Avala algorithm for larger deployment problems.

	100 comps 10 hosts 1 iteration	200 comps 20 hosts 1 iteration	1000 comps 100 hosts 1 iteration	100 comps 40 hosts 1 iteration	30 comps 7 hosts 1 iteration	300 comps 70 hosts 30 iterations
Unbiased maximum	0.580	0.562	0.503	0.534	0.602	0.520
Unbiased average	0.521	0.535	0.502	0.527	0.512	0.508
Biased maximum	0.696	0.691	0.527	0.590	0.828	0.610
Biased average	0.574	0.564	0.506	0.539	0.610	0.532
Avala	***0.787***	***.780***	***0.576***	***0.704***	***0.906***	***0.680***
% improvement over the unbiased average[a]	51.1	31.2	14.7	33.6	77.0	33.9

a. calculated as 100% * (Avala – unbiased average) / unbiased average

The following observations have further increased our confidence that Avala is finding nearly-optimal solutions for large systems: (1) for small systems (Table 1) the *unbiased average* was always very close to the *exact average*, denoting that the *unbiased average* precisely calculates the most likely availability; (2) the average improvement over the *unbiased average* for both small and large systems was quite similar (e.g., note the rightmost, i.e., benchmark columns of Tables 1 and 2); and (3) Avala's results for small systems were at least 90% of the optimal.

6.4 Calibrating the Avala Algorithm

As described in Section 5, Avala can be fine-tuned by assigning different values to the calibration factors *a*, *b*, *d*, and *e*. These factors denote the level of contribution of different parameters (link reliability, frequency of interaction, and memory of hosts and components) to the selection of the "best" host or "best" component. There are at least three different possibilities for selecting these factors: (1) predefined, constant values; (2) values selected and varied by a human user; or (3) automatically calculated values. We have implemented a generation facility for these factors that has been demonstrated experimentally to be quite effective. We have observed that, with the increase of the ratio of average host memory to average component memory, better results are obtained if more emphasis is placed on memory factors (i.e., increasing *b* and *e*) than on frequency and reliability factors (i.e., decreasing *a* and *d*). Experimentally we have obtained the best results for systems where the number of hosts is smaller than the number of components (i.e., $k<n$), calculating the calibration factors as:

$b = e = 0.1 * (\text{average host memory} * k) / (\text{average comp memory} * n)$ and $a = d = 1-b$

The benchmarks shown in Tables 1 and 2 are obtained using the above formulas for the calibration factors. Table 3 shows the benchmark data for the calibration factors using four different, randomly generated systems with varying numbers of components and hosts, and varying ranges for host and component memory. The *"Auto"* value cor-

responds to the factors calculated using the above formula, while the remaining rows of the table correspond to manually assigned factors. The resulting availability of automatically generated factors was within 1% of the best availability obtained with any other combination of factors.

7 Discussion

Here we discuss the characteristics as well as current limitations of the Avala algorithm, and suggest possible directions for addressing these limitations.

Table 3: Comparing the performance of Avala for different values of calibration factors, including their automatic generation.

Value for factors *a* and *c*	Value for factors *b* and *e*	100 comps 10 hosts avg host mem=85 avg comp mem=5	100 comps 10 hosts avg host mem=165 avg comp mem=11	15 comps 5 hosts avg host mem=60 avg comp mem=11	15 comps 5 hosts avg host mem=70 avg comp mem=11
0.9	0.1	0.739	0.759	0.75	0.85
0.8	0.2	0.745	0.775	0.772	0.86
0.7	0.3	0.739	0.769	0.773	0.858
0.6	0.4	0.737	0.757	0.772	0.856
0.5	0.5	0.732	0.738	0.74	0.853
0.4	0.6	0.722	0.728	0.734	0.837
0.3	0.7	0.699	0.717	0.706	0.837
0.2	0.8	0.672	0.706	0.642	0.837
0.1	0.9	0.66	0.701	0.622	0.832
Auto	Auto	0.744	0.772	0.772	0.861

7.1 Interaction Latency

For certain distributed systems, availability may not be the only, or the most crucial property. In fact, networked systems have traditionally focused on minimizing communication latencies as a key goal. Latency is commonly defined as the time taken to deliver a data packet from the source to the receiver [5]. While minimizing latency was not our primary goal in developing Avala, the algorithm's objective does naturally result in significant reductions of component communication latencies. The reason for this is two-fold. First, by increasing the overall system availability, some interactions that could not be successfully completed before now can be, thereby effectively reducing their latency from infinity to some finite time. Secondly, by employing the strategy of deploying frequently interacting components on the same host whenever possible, the latencies of those components' interactions are significantly reduced.

In order to compare the average interaction latency of a system's initial deployment to the deployment produced by Avala, we would have to average over all interaction latencies in the system in both deployments. Since in both cases there may be interactions that do not complete successfully due to network disconnections, those interaction latencies will be infinite, thus preventing us from comparing the average latency of the two deployments. For this reason, we will assume that network reliability of all host-to-host links is 1, i.e., that each component interaction successfully completes.

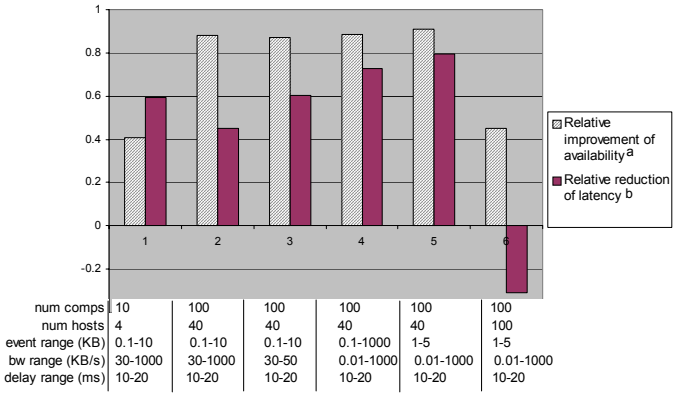
Latency of a single interaction depends on the following parameters: (1) startup latency, which is the constant communication overhead incurred in sending a zero length message [5], (2) network bandwidth of a link through which the interaction is performed, and (3) the size of message exchanged. To calculate the average latency in a given system, we use the following formula:

$$avgLatency = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}\left(freq(c_i,c_j)*\left(delay(f(c_i),f(c_j))+\frac{evt\_size(c_i,c_j)}{bw(f(c_i),f(c_j))}\right)\right)}{\sum_{i=1}^{n}\sum_{j=1}^{n}freq(c_i,c_j)}$$

where *delay* represents the startup latency of a given network link between two

hosts.[1] We assume that the latency of interaction for two components deployed onto the same host is zero (i.,e., *delay (h,h) =0* and *bw(h,h)=* ∞).

We have performed a series of benchmark tests to quantify the effect of Avala's results on average component interaction latency. To that end, we have extended our DeSi environment with random generation of startup latencies within a specified range, and automated calculation of average latencies for both a system's initial deployment and the deployment calculated by Avala. Figure 4. shows the results of these benchmarks. In most cases, redeployments produced by Avala reduced the average interaction latency by 40% - 80%.



a. calculated as (Avala_availability – initial_availability) /

Figure 4. The effect of redeployment calculated by Avala on average interaction latency. Each result was obtained by averaging over 20 different, randomly generated redeployment problems.

Avala, however, does not *guarantee* interaction latency reduction. In extreme cases, where each host's available memory is limited such that only a very small number of components can be deployed onto the host, the benefit of co-locating components cannot be leveraged. This case is illustrated in last column of Figure 4., where each host could only contain a single component due to memory constraints. Furthermore, since we are not assuming correlation between network reliability and bandwidth (e.g., a highly reliable link may have low bandwidth and vice versa), in some cases Avala may suggest deploying of components between hosts with high reliability and low bandwidth links, thus resulting in increased latency. One way to address this situation is to include bandwidth and event size as selection parameters for the "best" host and "best" component in Avala. We are currently implementing and evaluating this solution and its impact on both system availability and interaction latency.

7.2 Including the Constraints on Component Location

The benchmark results from Section 6.3 assess Avala's performance without using the *loc* and *colloc* constraints. We have also tested Avala with these constraints and have observed that, by introducing a significant number of constraints, the obtained availability starts to decrease. This is primarily due to the fact that the *loc* and *colloc* constraints will render invalid some deployments with otherwise high availabilities. For cases where either the size of the original problem or the reduction in the exact algorithm's complexity induced by the *loc* and *colloc* constraints enable us to invoke the exact algorithm, we have observed that the difference between the *exact maximum* and the availability produced by Avala actually decreases. The reason for this is that, as

1. Recall Section 2.2 for definitions of *evt_size* and *bw*.

the system becomes more constrained in terms of component location and collocation, the probability that Avala will divert significantly from the exact solution lessens.

7.3 Reducing the Memory Difference

By reducing the total available memory for hosts and/or increasing the total required memory for components, both the number of valid deployments and the system availability decrease. Again, this is due to the fact that a large number of deployments with otherwise high availabilities become invalid. In Tables 1 and 2 the highlighted columns are illustrative examples that correspond to these types of situations. For the system shown in Table 1, the total available memory for hosts was only 6% greater than the total required memory for components, resulting in 980 valid (out of over 1,000,000 possible) deployments. The relative improvement over the *unbiased average* was 11%, which was substantially lower than in other, less memory constrained systems. At the same time, the achieved availability was still more than 90% of the optimal availability. A similar situation can be observed in Table 2, although in that case the only available comparisons are to the *unbiased* and *biased averages*.

If the reduction of the total available memory for hosts and/or increase in the total required memory for components results in a very small number of valid deployments, our algorithm does not always find a valid deployment. The reason is that Avala initially assigns the component with the highest *initCompRank* to the host with the highest *initHostRank*. If this assignment leads to an invalid solution due to the limited available memory (e.g., just by assigning that component to that host the remaining components cannot be assigned), then our algorithm does not find a valid deployment.

One way to address this situation would be to detect cases when it occurs and try a different initial assignment. The number of different initial assignments is $k*n$, thus increasing the algorithm's complexity to $O(k*n^4)$. However, this still does not guarantee that the algorithm would find a valid deployment. We plan to assess this solution and possibly use additional backtracking techniques to address this limitation of Avala.

8 Conclusions and Future Work

As the distribution, decentralization, and mobility of computing environments grow, so does the probability that (parts of) those environments will need to operate in the face of network disconnections. Our research is guided by the observation that, in these environments, a key determinant of the system's ability to effectively deal with network disconnections is finding the appropriate *deployment architecture*. While the redeployment problem has been identified in the existing literature, its inherent complexity has either been ignored [1], thus making it infeasible for any realistic system, or highly restricted [6], thus reducing the solution's usefulness.

This paper has presented Avala, an efficient algorithm for improving a distributed, component-based system's availability via redeployment. Avala is part of an integrated solution to increasing a system's availability [14,15,17,12]. It has been thoroughly assessed via a series of benchmarks. In addition to significantly improving system availability Avala, in general, also reduces the overall interaction latency in the system. While our experience thus far has been very positive, a number of pertinent questions remain unexplored. Our future work will span issues such as (1) addressing situations in which the system constraints highly restrict the solution space, and (2) expanding our solution to include additional system parameters (e.g., battery power, display size,

system software available on a given host, and so on). These issues represent but a small subset of related concerns that are emerging in the domain of distributed, mobile computation and that will increasingly shape the software development of the future.

9 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Numbers CCR-9985441 and ITR-0312780. Effort also partially supported by the Jet Propulsion Laboratory.

10 References

[1] M. C. Bastarrica, et al. A Binary Integer Programming Model for Optimal Object Distribution. *2nd Int'l. Conf. on Principles of Distributed Systems*, Amiens, France, Dec. 1998.
[2] T. H. Cormen, et. al. Introduction to Algorithms. *MIT Press*, Cambridge, MA, 1990.
[3] A. Fuggetta, et. al. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 1998.
[4] D. Garlan, et al. Using Gauges for Architecture-Based Monitoring and Adaptation. *Working Conf. on Complex and Dynamic Systems Arch.,* Brisbane, Australia, Dec. 2001.
[5] http://www.epcc.ed.ac.uk/HPCinfo/glossary.html
[6] G. Hunt and M. Scott. The Coign Automatic Distributed Partitioning System. *3rd Symposium on Operating System Design and Implementation*, New Orleans, LA, Feb. 1999.
[7] IEEE Standard Computer Dictionary: *IEEE Standard Computer Glossaries.* New York, NY: 1990.
[8] T. Kichkaylo et al. Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques. *Int'l. Parallel and Distributed Processing Symposium*. April 2003.
[9] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, vol. 10, no. 1, February 1992.
[10] G. H. Kuenning and G. J. Popek. Automated Hoarding for Mobile Computers. Proc. of the *16th ACM Symp. on Operating Systems Principles,* St. Malo, France, October, 1997.
[11] S. Malek, et. al. A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems. In Proc. of *the 3rd Int. Working Conference on Component Deployment (CD 2005)*, Grenoble, France, Nov. 2005.
[12] S. Malek, et. al. Prism-MW: A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems. *IEEE Trans. on Software Engineering.* Vol. 31, No. 3, March 2005.
[13] N. Medvidovic, et al. Software Architectural Support for Handheld Computing. *IEEE Computer*, September 2003.
[14] M. Mikic-Rakic and N.Medvidovic. Software Architectural Support for Disconnected Operation in Highly Distributed Environments. *CBSE7*, Edinburgh, UK, May 2004.
[15] M. Mikic-Rakic, et. al. A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings. *2nd International Working Conference on Component Deployment (CD 2004)*, Edinburgh, UK, May 2004.
[16] M. Mikic-Rakic, et. al. Improving Availability in Large, Distributed, Component-Based Systems via Redeployment. Technical Report *USC-CSE-2003-515*, 2003.
[17] M. Mikic-Rakic and N. Medvidovic. Support for Disconnected Operation via Architectural Self-Reconfiguration. *Int. Conference on Autonomic Computing (ICAC'04)*, New York, May 2004.
[18] M. Mikic-Rakic and N. Medvidovic. Toward a Framework for Classifying Disconnected Operation Techniques. *ICSE WADS*, Portland, OR, May 2003.
[19] M. Mikic-Rakic and N. Medvidovic. Software Architectural Support for Disconnected Operation in Highly Distributed Environments. *Tech. Report*, USC-CSE-2003-506, 2003.
[20] Multi Router Traffic Grapher. http://scorpion77.cjb.net/mrtg/
[21] P. Oreizy et al. Architecture-Based run-time Software Evolution. *ICSE'98*, Japan, April 1998.
[22] Y. Weinsberg, and I. Ben-Shaul. A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices. *ICSE 2002*, Orlando, FL.
[23] J. Weissman. Fault-Tolerant Wide-Area Parallel Computing. *IPDPS 2000 Workshop*, Cancun, Mexico, May 2000.
[24] Y. Zhang, et.al. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. *Technical Report,* AT&T Center for Internet Research at ICSI, May 2000.