A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems

Sam Malek[1,3], Marija Mikic-Rakic[2], 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. Determining a software system's deployment that will maximize its availability is an exponentially complex problem. Although several polynomial-time approximative techniques have been developed recently, these techniques rely on the assumption that the system's deployment architecture and its properties are accessible from a central location. For these reasons, the existing techniques are not applicable to an emerging class of decentralized systems marked by the limited system wide knowledge and lack of centralized control. In this paper we present an approximative solution for the redeployment problem that is suitable for decentralized systems and assess its performance.

1 Introduction

Highly distributed and mobile systems are challenged by the problem of *disconnected operation* [25], where the system must continue functioning in the temporary absence of the network. Disconnected operation forces systems executing on each network host to temporarily operate independently from other hosts. 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 [8]. In the context of highly distributed, mobile environments, where the most common cause of (partial) system inaccessibility is network failure [24], 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.

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., network reliability) 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.

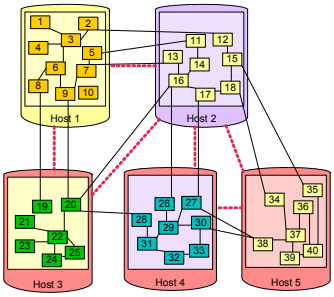

Figure 1. A sample deployment architecture with five hosts and 40 components.

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 [2] to facilitate redeployment, and dynamic system manipulation [20] to effect the redeployment once the components are migrated to the appropriate hosts. However, one of the critical difficulties 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.

This paper accompanies our work on providing a centralized solution, which is complementary to this paper and requires global knowledge of system parameters and global control of the system's redeployment [18]. Therefore, the centralized solution assumes the existence of a central host that has reliable access to every other host in the system. This assumption has made the centralized solution inapplicable to a wide range of distributed systems (e.g., ad-hoc mobile networks) where such a reliable centralized host does not exist.

In this paper we present an approximative algorithm for increasing a system's availability that scales to the exponentially complex nature of this problem. The algorithm, called DecAp, is decentralized and does not require global knowledge of system properties. We provide a detailed assessment of DecAp's performance through its comparison against several centralized algorithms. We leverage our deployment exploration environment, called DeSi [17], in performing DecAp's performance assessment. DeSi supports quantitative assessment and comparison of different redeployment algorithms as well as active visualization of a system's deployment 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. Section 4 describes the DecAp algorithm and discusses its complexity. Section 5 discusses DecAp's behavior. Section 6 presents our approach for evaluating DecAp and the results of its assessment. The paper concludes with a discussion of future work.

2 The Redeployment Problem

2.1 Problem Definition

We describe a distributed system as (1) a set of *n* components with their properties, (2) a set of *k* hosts with their properties, (3) a set of constraints that a valid deployment architecture must satisfy, (4) the system's initial deployment as a mapping of components to hosts, and (5) a set of system properties that are "visible" from a given host. Figure 2. shows a formal model that captures the above system properties and constraints.

(1) A set C of n components ($n = |C|$) and two functions

$freq : C \times C \rightarrow \Re$ and $mem_{comp} : C \rightarrow \Re$

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

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

(2) A set H of k hardware nodes ($k = |H|$) and two functions

$rel : H \times H \rightarrow \Re$ and $mem_{host} : H \rightarrow \Re$

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

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

(3) Two functions that restrict locations of software components

$loc : C \times H \rightarrow \{0,1\}$ and $colloc : C \times C \rightarrow \{-1,0,1\}$

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

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

(4) A relation $dep : H \rightarrow \mathcal{P}(C)$ where $c_k \in dep(h_i)$ iff c_k is deployed on h_i

(5) A function $aware : H \times H \rightarrow \{0,1\}$

$$aware(h_i, h_j) = \left\{ \begin{array}{l} 1 \quad if\ h_i\ and\ h_j\ \text{have the following information:} \\ \forall c_i \in dep(h_i), \forall c_j \in dep(h_j), \forall c_k \in C, \forall h_k \in H \\ mem_{host}(h_i),\ mem_{host}(h_j),\ dep(h_i),\ dep(h_j), \\ rel(h_i, h_j),\ mem_{comp}(c_i),\ mem_{comp}(c_j),\ freq(c_i, c_k), \\ freq(c_j, c_k),\ loc(c_i, h_k),\ loc(c_j, h_k), \\ colloc(c_i, c_k),\ colloc(c_j, c_k). \\ 0 \quad \text{if } h_i \text{ and } h_j \text{ have no information of each other} \end{array} \right\}$$

and a relation $dom : H \rightarrow \mathcal{P}(H)$, where $h_k \in dom(h_i)$ iff $aware(h_k, h_i) = 1$

Figure 2. Formal redeployment model.

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* function. 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* function. Using the *loc* function, 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* function, constraints on collocation of components can be specified. The relation *dep* denotes the current deployment of the system's components on hosts.

The function *aware* and the relation *dom* model the system's decentralized nature. Function *aware* denotes whether two hosts have access to each other's properties and the properties of components that reside on them. Relation *dom* denotes the "domain" of a host h_i, which is the set of all hosts of which h_i is aware. A host's domain corresponds to the host's extent of knowledge about the overall system's parameters. For example, in the centralized approach to the redeployment problem discussed above, the assumption is that at least one host's domain is the entire set of hosts *H*.

Figure 3. shows a formal definition of the problem we are solving. The criterion function A 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 stated conditions: (1) the sum of memories of the components that are deployed onto a given host may not exceed the available memory on that host; (2) a component may only be deployed onto a host that belongs to the set of allowed hosts for that component, specified via the *loc* function; and (3) two components must be deployed onto the same host (or on different hosts) if required by the *colloc* function.

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

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

is maximized, and the following three conditions are satisfied:

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

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

(3) $\forall k \in [1,n] \quad \forall l \in [1,n]$

if $(colloc(c_k,c_l) = 1) \Rightarrow (f(c_k) = f(c_l))$

if $(colloc(c_k,c_l) = -1) \Rightarrow (f(c_k) \neq f(c_l))$

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 3. Formal statement of problem definition

2.2 Assumptions

The problem defined in Figure 3. is an instance of the more general redeployment problem, described in [18]. 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 [18] 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.

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* function can be changed to correspond to the product of interaction frequency and importance, 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 reasonably 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,

1. We do not require that system parameters be constant during *T*, but assume that each parameter can be approximated with its average over the period *T*, with an error no greater than a given threshold ε [19].

the system's parameters would be changing too frequently and the system would undergo continuous redeployments to improve its availability.

Finally, our approach is based on the assumption that two hosts that are aware of each other will be able to reliably exchange the "meta-level" information (detailed in Section 4) required for the correct functioning of the redeployment algorithm. This can be ensured by employing existing techniques, e.g., delivery guarantee mechanisms [14], or gossip-based protocols [3]. While such techniques may also be used to improve the availability of the system itself, employing them for all *application-level* information exchange will typically be too expensive.

3 Related work

In this section we present a brief overview of centralized redeployment approaches. We also provide an overview of most commonly used decentralized cooperative algorithms.

3.1 Centralized Deployment Approaches

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 [7] 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. [11], 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.

Finally, we have developed several algorithms for the centralized version of the redeployment problem [18]. In section 6.2, we briefly describe these algorithms, as they will be used to assess the performance of DecAp.

3.2 Decentralized Cooperative Algorithms

Decentralized cooperative algorithms have been used in distributed systems to achieve higher degrees of fault-tolerance, load balancing, and performance. The emergence of decentralized environments, such as mobile ad-hoc networks and peer-to-peer sensor networks has required decentralized algorithms to enable autonomous agents to coordinate their interactions, make local decisions based on limited information, and cooperate with other agents to achieve the overall system goals. We discuss some of the most common decentralized cooperative approaches.

Voting [12] is a method for coordinating distributed systems. A set of distributed processors works independently on the same task, and then votes on their results to select one correct answer. Decentralized voting [6,10] increases the fault-tolerance in a distributed system by using replicated voters to independently determine the majority result, rather than relying on a central server to tally the results. In the context of the redeployment problem, if each host independently calculates the system's redeployment based on limited information, voting techniques could be employed to decide which one of the redeployments should be effected.

Token Ring [9] is a classic solution to distributed mutual exclusion problems. All hosts are arranged into a set of logical structures called rings. All communication occurs along the channels that define a ring. One or more tokens circulate around the ring. To use a shared resource, a host needs to acquire a token. When the host is finished, it passes the token to the next host. The token ring technique can be used in the context of the decentralized redeployment problem to control the simultaneous component migrations in the system.

Market-Based [13,23] approaches are derived from economics concepts such as trading and auctioning. The most popular market-based solution is the *auction* algorithm, in which each auctioneer agent conducts auctions to sell some items (i.e., provided services or resources) by broadcasting an auction initiation message. A bidding agent interested in an auctioned item sends a bid to the auctioneer agent. The bid is typically calculated using a utility function that determines the bidding agent's interest in the auctioned item. The auctioneer agent determines the winner (typically the highest bidder) and awards it the item. As will be detailed in section 4, DecAp leverages the market-based approach for improving the system's availability.

4 The DecAp Algorithm

DecAp is a decentralized, collaborative auctioning algorithm for improving system-wide availability. Each host in DecAp contains a single autonomous agent. These agents collaborate to improve the overall system's availability. Each agent has access to the monitoring data within its domain of awareness (recall Figure 2.). An agent exchanges messages with other agents that are members of its host domain.

The auctioned items in DecAp are software components. For a component to be ready for auctioning, its relevant parameters must be stable [19]. An agent plays two roles during the redeployment process: (1) auctioneer, in which the agent conducts the auction of its local components, and (2) bidder, in which the agent bids on components auctioned by a remote agent. DecAp extends the classic auction algorithm in two ways: (1) an auctioneer is allowed to participate in auctions it conducts, by setting the minimum bid for the auctioned component; and (2) the auctioneer may adjust the received bids.

To participate in an auction conducted on host h_a, a bidder agent has to reside on one of the hosts that are members of h_a's domain. Each agent can be in one of the following three states: *auctioning*, *bidding*, or *free*. The auctioning process for a single component is as follows. First, the auctioneer announces an auction of a local component c_a. It then receives all the bids from bidders within its domain. Finally, the auctioneer determines the "winner", i.e., the location for c_a within $dom(h_a)$ that results in highest availability. To ensure that the winner is correctly determined, agents participating in this auction cannot participate in other auctions at the same time.

As a result of a single auction, a component can move only to one of the hosts that are inside the domain of the component's auctioneer host. For this reason, multiple auctions of a single component may be required before the "sweet spot" for that component in the given distributed system is found. A component's sweet spot is its deployment location that does not change as a result of future auctions for that component. This is known as the Nash Equilibrium State in market-based literature [13].

DecAp's auctioneer and bidder algorithms use the following two functions:

1. the *contribution* of component c_x to the overall availability of the domain of host h_x when c_x is deployed on h_x, defined as follows:
$$contribution(c_x, h_x) = \sum_{h_i \in dom(h_x)} \sum_{c_j \in dep(h_i)} (rel(h_x, h_i) * freq(c_x, c_j))$$
2. the available memory, (i.e., *freeMemory*) on a given host h_x, defined as follows:
$$freeMemory(h_x) = mem_{host}(h_x) - \sum_{c_i \in dep(h_x)} mem_{comp}(c_i)$$

Below we describe both the auctioneer's and the bidder's algorithms and how they are coordinated.

4.1 Auctioneer's Algorithm

The auctioneer's algorithm, performed on auctioneer's host h_a for one of its software components c_a (i.e., $c_a \in dep(h_a)$), consists of the following eight steps, repeating the steps for each component on h_a:

1. If c_a is ready to be auctioned, calculate the minimum bid for c_a as follows: $minBid(c_a) = contribution(c_a, h_a)$
2. If h_a's *state* is *free*, change it to *auctioning*, send the AUCTION INTENT message to all hosts in $dom(h_a)$, and proceed to step 3. Otherwise, wait for a given time interval and repeat step 2.
3. If all hosts in $dom(h_a)$ respond with an AUCTION ACCEPT message before the specified time-out, continue to step 4. Otherwise, send AUCTION CANCEL message to all hosts in $dom(h_a)$, set h_a's *state* to *free*, wait for random time interval, and go back to step 2.
4. Broadcast an AUCTION START message to every host in $dom(h_a)$. Include the *minBid* in the message. The *minBid* sets up a threshold for an acceptable bid. It is used by the bidders to determine whether they qualify to participate in the auction or not.
5. When the bids from all the hosts in $dom(h_a)$ are received, or a time-out occurs, adjust the bids from the hosts that do not have enough memory for the auctioned component. When a bidding host does not have enough memory for component c_a, it needs to trade c_a with one of its local components. As will be detailed in section 4.2, each host h_b for which $freeMemory(h_b) < mem_{comp}(c_a)$, in addition to the bid, sends a set of "tradable" components' identifiers $T \subseteq dep(h_b)$ and their contributions (i.e., $\forall c_x \in T \,|\, contribution(c_x, h_b)$). For each host h_b, the auctioneer determines the best candidate component for trade c_t, as a component whose migration from h_b to h_a will have the smallest negative impact on the availability, as follows:
$$c_t = \min\left\langle \forall c_x \in T \,|\, contribution(c_x, h_b) - contribution(c_x, h_a) \right\rangle$$

Then, the auctioneer recalculates the bid from host h_b to adjust for the effect of the

trade, as follows:

$$bid(c_a, h_b) =$$
$$bid(c_a, h_b) - (freq(c_t, c_a) - rel(h_a, h_b) * freq(c_t, c_a)) - (contribution(c_t, h_b) - contribution(c_t, h_a))$$

When adjusting the bids for all the hosts that do not have enough memory is complete, go to step 6.

6. Find the winner host h_w by selecting the highest bidder. If $bid(c_a,h_w) > minBid$, continue to step 7. Otherwise, c_a remains deployed on h_a; skip to step 8.
7. If h_w has enough memory (i.e. $freeMemory(h_w) > mem_{comp}(c_a)$), migrate c_a to h_w. Otherwise, perform the trade by migrating c_a to h_w and migrating c_t to h_a.
8. Broadcast an AUCTION TERMINATION message to every host in $dom(h_a)$ to denote the completion of this auction. Set h_a's *state* to *free*.

4.2 Bidder's Algorithm

The bidder's algorithm, where $h_b \in dom(h_a)$ is the bidder host, consists of the following eight steps:

1. When an AUCTION INTENT message arrives, if h_b's *state* is *free*, send the AUCTION ACCEPT message to h_a, set the *state* to *bidding*, and continue to step 2. Otherwise, send the AUCTION REJECT message to h_a.
2. If an AUCTION CANCEL message arrives, set the *state* to *free,* and go back to step 1. If the AUCTION START message arrives from h_a, calculate the bid for c_a as the contribution of c_a to the availability of $dom(h_b)$ if c_a were to be deployed on h_b: $bid(c_a, h_b) = contribution(c_a, h_b)$
3. If $bid(c_a,h_b) < minBid$, h_b does not qualify to place a bid on c_a, skip to step 8. Otherwise create the bid message by including the $bid(c_a,h_b)$. Proceed to step 4.
4. If h_b has enough free memory for c_a (i.e. $freeMemory(h_b) > mem_{comp}(c_a)$), proceed to step 7.
5. Since h_b does not have enough memory for c_a, find the set $T \subseteq dep(h_b)$ of "tradable" components. A component is tradable when it has the adequate memory size for the trade as follows:

$$T = \{\forall c_x \in dep(h_b) \mid mem_{comp}(c_a) \leq$$
$$(mem_{comp}(c_x) + freeMemory(\mathrm{h_b})) \wedge mem_{comp}(c_x) \leq (mem_{comp}(c_a) + freeMemory(\mathrm{h_a}))\}$$

6. If T is not empty, append to the bid message both the identifiers of all components $c_x \in T$ and their contributions, $contribution(c_x,h_b)$, and proceed to step 7. Otherwise, when T is empty, a tradable component does not exist and component c_a cannot be deployed onto h_b; skip to step 8.
7. Place the bid by sending the bid reply message to h_a.
8. Upon arrival of the AUCTION TERMINATION message, set h_b's state to *free*.

4.3 Analysis of the Two Algorithms

To ensure that an agent participates in a single auction at a time, we employed a distributed locking mechanism using the *state* variable for each agent as described in steps 2, 3, and 8 of the auctioneer's algorithm, and steps 1, 2, and 8 of the bidder's algorithm. To avoid deadlocks and starvation, each auctioneer waits a random interval of time before the next attempt at starting an auction.

The worst-case time complexity analysis for each of the two algorithms is given below (where k is the number of hosts and n is the number of components). Note that the analysis of agent synchronization time complexity is not provided, since we adopted a well-known distributed locking technique, whose complexity analysis is provided in [22]. We also do not analyze the time complexity of performing the migration of components between hosts, since a detailed analysis is provided in [19].

$O(auctioneer) = O(step\ 1) + O(step\ 5) + O(step\ 6) = O(k*n) + O(n*k*n) + O(k) = O(k*n^2)$

$O(bidder) = O(step\ 2) + O(step\ 5) = O(k*n) + O(n) = O(k*n)$

Finally, the auctioneer's algorithm will be executed several times for each software component. Some of these auctions may occur simultaneously within the entire system, depending on the number of components on each host and the number of hosts within each host's domain. In the worst case (e.g., domain of each host is the entire set of hosts H), the auctioneer's algorithm executes in a sequential manner for each component, resulting in the total complexity of DecAp to be $n*O(auctioneer) = O(k*n^3)$.

5 Discussion

Below we discuss the salient aspects of DecAp's behavior and performance in more detail.

Algorithm's Guarantee to Find a Solution. In [18] we identified situations where the centralized algorithms do not always find a solution (e.g., if the total number of deployments that satisfy all the constraints from Figure 3. is very small). In such situations, DecAp can still find an improved deployment, since it focuses on localized, incremental improvement to the overall availability.

Algorithm's Convergence. DecAp performs a redeployment of components only if it results in the overall system's availability increase. For this reason, each auction guarantees that the system's availability will either increase or remain the same (if the auctioned component remains on the auctioneer host). As will be illustrated in section 6, the algorithm typically converges after only a few auctions for each component, i.e., subsequent auctions do not change the deployment architecture of the system. As soon as the given host becomes the "sweet spot" for all of its components, the auctioneer algorithm on that host assumes the algorithm's convergence with a certain degree of confidence, and extends the period of time before attempting a new auction (i.e., the host's dormant time). If during subsequent auctions the host remains the "sweet spot" for its components, its degree of confidence, and thus the period of dormancy, increase.

Algorithm's Sensitivity to the Level of Awareness. DecAp provides a flexible approach for capturing the level of awareness present at each node, through careful definition of the *aware* function and *dom* relation in our model. DecAp's model does not make any assumptions about what constitutes awareness among two hosts (i.e., when $aware(h_i,h_j)=1$). We simply set a given host's domain (i.e., the *dom* relation) to the set of all the hosts of which it is *aware*. The model can then be instantiated with an implementation-level definition of awareness. Some commonly used policies in determining aware hosts are: directly connected hosts, proximity of hosts, number of node hops, bandwidth or signal strength, and reliability of links. Figure 4. illustrates the effect of using different policies for determining host awareness. While our algorithm is inde-

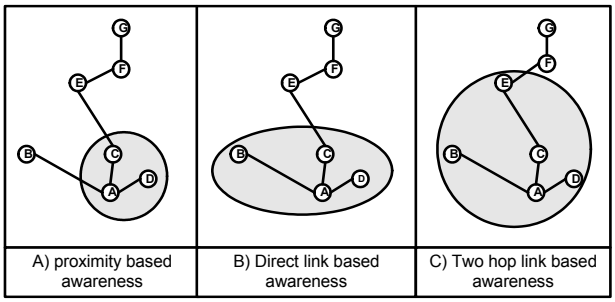

Figure 4. Domain of host A with different policies for determining host awareness.

pendent of the policy that constitutes host awareness, the performance of the algorithm is significantly affected by the level of awareness present at each host. We will demonstrate the sensitivity of our algorithm to the level of awareness in the next section.

Location Constraints. In section 2.2 we discussed how using the *loc* and *colloc* functions can be leveraged to capture constraints other than memory. For clarity the algorithm presented in section 4 did not explicitly describe how the location constraints remain satisfied throughout the algorithm's execution. The constraint imposed by the *loc* function is enforced by inviting the hosts to participate in an auction only if they satisfy the *loc* constraint.The constraint imposed by the *colloc* function is enforced as follows: (1) when a component cannot be on the same host as the auctioned component, the auctioneer simply does not invite the host that contains that component to the auction, and (2) when two component have to be on the same host, the components are merged into a single virtual component and therefore always auctioned at the same time. Also note that through the use of *loc* and *colloc*, the complexity of the algorithm is reduced proportionally to the extent of the constraints imposed by the two functions in the given system [18].

Consideration of Additional System Properties. For certain distributed systems, availability may not be the only, or the most crucial property. For example, traditional networked systems have extensively focused on minimizing communication latencies. While minimizing latency was not our primary goal in developing DecAp, we should point out that the algorithm's objective (deploying frequently interacting components on the same host or on hosts with reliable network links) does naturally result in significant reductions of component communication latencies. We are currently trying to quantify the exact impact of DecAp on latency. Another relevant issue is the inclusion of network bandwidth in the system model, and the resulting algorithm. As discussed in section 2.2, in certain situations the location and collocation constraints can be leveraged to capture additional system parameters, including network bandwidth. However, if bandwidth becomes a scarce resource in the system, it will need to be considered separately. Our experience with the centralized redeployment algorithms (see section 6.2) indicates that this parameter can be easily added to the system model and that the resulting change to the algorithms themselves is straightforward.

6 Evaluation

In this section we provide a description of our approach in evaluating the performance of DecAp. We also provide a detailed comparison of DecAp's performance against several centralized algorithms. Note that since DecAp is the first decentralized solution to the redeployment problem of which we are aware, we can only compare its performance against the existing centralized solutions.

6.1 DecAp's Implementation

In order to quickly assess the performance of DecAp on large numbers of redeployment problems, involving large numbers of software components and hardware hosts, we implemented a simulated version of DecAp that runs on a single physical host. The distribution aspect of DecAp is simulated through the use of multiple, autonomous agents. We simulated the decentralization aspect of DecAp through the use of multiple threads and limited visibility among agents. DecAp was implemented in Java and integrated with our deployment exploration environment DeSi. When DeSi's user interface invokes DecAp, a bootstrap thread instantiates an agent object for each host. Each agent class is composed of two inner classes: auctioneer class and bidder class. Both auctioneer and bidder classes have their own threads of execution, which are started once the corresponding agent class is instantiated. Agents in the same domain are given access to each other's class variables. In our implementation of DecAp, we used direct links to denote the awareness level of 1 (recall Figure 4.B). Subsequent levels of awareness correspond to the number of intermediate hosts between a pair of hosts (recall Figure 4.C). Auctioneer and bidder threads synchronize their interactions through message passing. A shared data structure that holds the current deployment of the system is updated as a result of each auction. DeSi's bootstrap class calculates the overall availability of the shared data structure in pre-specified time intervals. The algorithm terminates when the availabilities at two consecutive time intervals are the same, which indicates that the algorithm has converged to a solution.

6.2 Evaluation Criteria

In this section, we briefly describe three centralized algorithms we have developed previously for increasing a system's availability by calculating a new deployment architecture. A detailed explanation and evaluation of these algorithms is given in [18]. These algorithms provide the basis for evaluating DecAp.

Exact Algorithm. This algorithm tries every possible deployment, and selects the one that has maximum availability and satisfies the constraints posed by the memory and restrictions on software component locations (*exact maximum*). This algorithm also finds the average availability of all system deployments (*exact average*). The exact algorithm guarantees at least one optimal deployment (assuming that at least one deployment is possible). 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. For this reason, executing the exact algorithm is only feasible for very small systems.

Unbiased Stochastic Algorithm. This algorithm generates different deployments by randomly assigning each component to a single host from a set of component's allowable hosts. If the 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 deployment with the best availability is selected (*unbiased maximum*). The average availability of all valid deployments is also calculated (*unbiased average*). The complexity of this algorithm is $O(n^2)$. In [18] we have experimentally shown that *unbiased average* does not significantly deviate from the *exact average* and thus signifies the system's "most likely" availability.

Greedy Algorithm. This algorithm 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, by selecting the "best" host and "best" software component. Selecting the best hardware host is performed by choosing a host with the highest sum of network reliabilities with other hosts in the system, and the highest memory capacity. Similarly, selecting the best software component is performed by choosing the component with the highest frequency of interaction with other components in the system, and the lowest required memory. Once found, the best component is assigned to the best host, making certain that all the constraints are satisfied. The algorithm proceeds with searching for the next best component among the remaining components, until the best host is full. Next, the algorithm selects the best host among the remaining hosts. This process repeats until every component is assigned to a host. The availability of the resulting deployment (*greedy maximum*) is calculated. The complexity of this algorithm is $O(n^3)$ [18].

6.3 Evaluation Results

Table 1 provides the comparison of DecAp with the three centralized algorithms, in cases where the graph of hosts is fully connected (possibly via unreliable links). Columns 4 and 5 show the results of running the algorithms for 25 different redeployment problems and averaging the results using the benchmarking option of DeSi. DecAp provided at least 40% improvement over the system's "most likely" deployment. On average, DecAp produced results that were better than the centralized algorithms' results. However, in certain situations the performance of DecAp could suffer, due to its reliance on the initial deployment. For example, in situations where some of the "best" hosts (recall the above description of the greedy algorithm) in the system do not

Table 1: Comparison of DecAp's performance in deployment architectures with fully connected graph of hosts.

		1	2	3	4	5
		10 comps 4 hosts 1 problem	50 comps 15 hosts 1 problem	250 comps 50 hosts 1problem	10 comps 4 hosts 25 problems	50 comps 15 hosts 25 problems
1	Exact maximum	0.816	infeasible	infeasible	0.792	infeasible
2	Exact average	0.553	infeasible	infeasible	0.525	infeasible
3	Unbiased maximum	0.756	0.611	0.512	0.699	0.544
4	Unbiased average	0.550	0.558	0.469	0.525	0.508
5	Greedy maximum	0.807	0.734	0.641	0.720	0.729
6	*DecAp* *Awareness level = 1*	*0.790*	*0.759*	*0.653*	*0.756*	*0.764*
7	% improvement over the unbiased average[a]	43	36	39	44	50

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

Table 2: .Comparison of DecAp's performance in deployment architectures with varying levels of disconnected links among hosts.

		1	2	3	4	5	6
		50 comps 15 hosts 20% of links missing	50 comps 15 hosts 50% of links missing	50 comps 15 hosts 80% of links missing	100 comps 25 hosts 30% of links missing	100 comps 25 hosts 60% of links missing	100 comps 25 hosts 90% of links missing
1	Original availability	0.427	0.265	0.176	0.385	0.227	0.06
2	Unbiased maximum	0.442	0.319	0.184	0.407	0.258	0.105
3	Unbiased average	0.442	0.284	0.146	0.375	0.219	0.084
4	Greedy maximum	0.604	0.530	0.339	0.590	0.411	0.283
5	DecAp Awareness level = 1	0.644	0.479	0.301	0.613	0.445	0.194
6	DecAp Awareness level = 2	0.747	0.582	0.349	0.618	0.455	0.250
7	DecAp Awareness level = 3	0.747	0.582	0.367	0.618	0.460	0.261
8	% improvement over original availability	74	119	108	60	102	335

have any components initially deployed on them, they may not ever be selected as the winners of any of the auctions.

Table 2 provides another comparison of DecAp with centralized algorithms in cases where the graph of hosts is not fully connected (each column is labelled with the percentage of missing host-to-host links). For each problem, the DecAp algorithm was executed three times with different levels of awareness. As the table indicates, the algorithm's performance is negatively affected by the decrease in host inter-connectivity. However, as long as the graph of hosts is connected, increasing the level of awareness improves DecAp's performance significantly. Columns 1-5 show such a scenario, where as a result of increasing the level of awareness, the algorithm outperforms even the centralized algorithms. Column 6 shows another scenario, where as a result of a very high percentage of missing links, "islands" of hosts (i.e. subsets of hosts that are not connected to each other) are created and DecAp is not able to outperform the greedy algorithm. Finally, row 8 shows that DecAp was able to improve the availability by at least 60% over the original availability in the case of a fairly connected architecture, and by at most 335% in the case of a fairly disconnected architecture.

Table 3: Demonstration of DecAp's convergence.

Iteration Number	10 comps 4 hosts 20% of links missing 1 level of awareness	50 comps 15 hosts 50% of links missing 1 level of awareness	100 comps 25 hosts 70% of links missing 1 level of awareness	250 comps 50 hosts 80% of links missing 2 levels of awareness
Initial Availability	0.450	0.254	0.174	0.099
1	0.776	0.423	0.312	0.219
2	0.881	0.483	0.334	0.231
3	0.910	0.500	0.342	0.243
4	0.933	0.503	0.350	0.248
5	0.974	0.519	0.354	0.250
6	0.974	0.529	0.360	0.253
7	0.974	0.529	0.360	0.253
% first iteration / final solution	79%	79%	86%	86%

Table 3 shows DecAp's convergence to a solution. Each iteration corresponds to the resulting availability of the overall system after auctioning each one of the components

exactly once. Note that the largest gain is achieved in the first iteration of the algorithm, which shows that by just auctioning each component once, we can get a solution that is at least 79% of the final solution. Also note that after the first iteration of the algorithm, most components have found a "sweet spot", which results in no further redeployment of those components. This contributes to the quick convergence of the algorithm, typically around the fifth or sixth iteration. For the largest problem (shown in the last column of Table 3), DecAp's execution time was 9.4s with the maximum auctioneer thread wait of 10ms. However, a variation of DecAp that used thread notification executed the same problem in 0.3s on a mid-range PC.[1]

7 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 addressed by our work 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 [7], thus reducing the solution's usefulness. Furthermore, the existing solutions are not applicable to an emerging class of decentralized systems marked by the limited system knowledge and lack of centralized control.

This paper has presented an efficient decentralized algorithm for improving a distributed, mobile, component-based system's availability via redeployment. The algorithm is currently being integrated into an existing middleware platform [15] with built-in capabilities for system monitoring and redeployment [19]. The algorithm has been thoroughly assessed via a series of benchmarks. While our experience thus far has been very positive, a number of pertinent questions remain unexplored. In addition to assessing the performance of DecAp in a truly distributed environment, our future work will span issues such as (1) extending the algorithm to identify "good" hosts in the system even when they initially do not have any deployed components, (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), and (3) leveraging techniques such as simulated annealing [21] to further improve the algorithm's performance. 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.

8 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.

---

1. Since we only wanted to illustrate the execution time of the algorithm's logic, and not that of agents' synchronization, to obtain this result we leveraged the thread notification technique instead of the random thread wait times described in Section 4. Note that employing thread notification is possible only in a single-processor simulation of the algorithm.

9 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. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, May 1998.
3. A. J. Ganesh, A. Kermarrec, L. Massoulie. Peer-to-Peer Membership Management for Gossip-Based Protocols, *IEEE Transactions on Computers*, Vol. 52, pp. 139-149, Feb. 2003.
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. D. K. Gifford, Weighted Voting for Replicated Data. In Proceedings of the *7th Symposium on Operating System Principles*, New York, 1979, pp. 150-162.
6. B. Hardekopf, et. al. A Decentralized Voting Algorithm for Increasing Dependability in Distributed Systems. *5th World Multi- Conference on Systemic, Cybernetics and Informatics (SCI2001),* 2001.
7. G. Hunt and M. Scott. The Coign Automatic Distributed Partitioning System. *3rd Symposium on Operating System Design and Implementation*, New Orleans, LA, Feb. 1999.
8. IEEE Standard Computer Dictionary: *A Compilation of IEEE Standard Computer Glossaries.* New York, NY: 1990.
9. W. Jia, J. Kaiser, E. Nett. An Efficient and Reliable Group Multicast Protocol. *Second International Symposium on Autonomous Decentralized Systems*. Phoenix, Arizona., April 1995.
10. B. Johnson. Design and Analysis of Fault Tolerant Digital Systems, *Addison-Wesley*, 1989.
11. T. Kichkaylo et al. Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques. *Int'l. Parallel and Distributed Processing Symposium*. April 2003.
12. R. Kieckhafer, C. Walter, A. Finn, P. Thambidurai. The MAFT Architecture for Distributed Fault Tolerance. *IEEE Transactions On Computers*, Vol. 37, No. 4, April 1988, pp. 398-405.
13. D. Kreps. Game Theory and Economic Modeling. *Clarendon Press*, Oxford, 1990.
14. E. A. Lee. Embedded software. *Advances in Computers, 56*, 2002.
15. S. Malek, M. Mikic-Rakic and N. Medvidovic. Prism-MW: A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems. *IEEE Trans. on Software Engineering.* Vol. 31, No. 3, March 2005.
16. N. Medvidovic, et. al. Software Architectural Support for Handheld Computing. *IEEE Computer*, September 2003.
17. 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.
18. M. Mikic-Rakic, et. al. Improving Availability in Large, Distributed, Component-Based Systems via Redeployment. In Proceeding of *the 3rd International Working Conference on Component Deployment (CD 2005)*, Grenoble, France, Nov. 2005.
19. M. Mikic-Rakic and N. Medvidovic. Software Architectural Support for Disconnected Operation in Highly Distributed Environments. *International Symposium on Component-based Software Engineering (CBSE7)*, Edinburgh, UK, May 2003.
20. P. Oreizy et al. Architecture-Based run-time Software Evolution. *ICSE'98*, Kyoto, Japan, April 1998.
21. S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. *Prentice Hall,* Englewood Cliffs, NJ, 1995.
22. A. Tanenbaum. Computer Networks. *Prentice Hall*, Englewood Cliffs, New Jersey.
23. C. A. Waldpurger, et. al. Spawn. A Distributed Computational Economy. *IEEE Trans. on Software Engineering*, February 1992
24. J. Weissman. Fault-Tolerant Wide-Area Parallel Computing. *IPDPS 2000 Workshop*, Cancun, Mexico, May 2000.
25. Y. Weinsberg, and I. Ben-Shaul. A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices. *ICSE 2002*, Orlando, FL.