Software Architectural Support for Disconnected Operation in Highly Distributed Environments

Marija Mikic-Rakic and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
{marija,neno}@usc.edu

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., 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 an automated, flexible, software architecture-based solution for disconnected operation that increases the availability of the system during disconnection. We provide a fast approximative solution for the exponentially complex redeployment problem, and assess its performance.

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. One can now envision a number of complex software development scenarios involving fleets of mobile devices used in environment monitoring, traffic management, damage surveys in times of natural disaster, and so on. Such scenarios present daunting technical challenges: effective understanding of software configurations; rapid composability and dynamic reconfigurability of software; mobility of hardware, data, and code; scalability to large amounts of data and numbers of devices; and heterogeneity of the software executing across devices. Furthermore, software often must execute on "small" devices, characterized by highly constrained resources such as limited power, low network bandwidth, slow CPU speed, limited memory, and patchy connectivity. We refer to the development of software systems in the described setting as *programming-in-the-small-and-many (Prism)*, in order to distinguish it from the commonly adopted software engineering paradigm of programming-in-the-large (PitL) [8].

Applications in the Prism setting are becoming highly distributed, decentralized, and mobile, and therefore highly dependent on the underlying network. Unfortunately, network connectivity failures are not rare: mobile devices face frequent and unpredictable (involuntary) connectivity losses due to their constant location change and lack of wireless network coverage; the costs of wireless connectivity often also induce user-initiated (voluntary) disconnection; and even the highly reliable WAN and LAN connectivity is unavailable 1.5% to 3.3% of the time [25].

For this reason, Prism systems are challenged by the problem of *disconnected operation* [24], 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 a degree to which the system suffers degradation or interruption in its service as a consequence of failures of one or more of its components [11]. In the context of Prism systems, where a most common failure is a network failure, we define availability as the ratio of the number of successfully completed inter-component interactions in the system to the total number of attempted interactions.

A key observation for systems executing in the Prism setting is that the distribution of software components onto hardware nodes (i.e., *deployment architecture*) greatly influences the system's availability in the face of connectivity losses. However, the parameters that influence the optimal distribution of a system 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.

There are several existing techniques that can support various subtasks of redeployment, such as monitoring [10] to assess hardware and software properties of interest, component migration [9] to facilitate redeployment, and dynamic system manipulation [19] to effect the redeployment once the components are migrated to the appropriate hosts. However, the critical difficulty lies in the fact that determining a software system's deployment that will maximize its availability is an exponentially complex problem. Existing approaches that recognize this (e.g., [2]) still assume that all system parameters are known beforehand and that infinite time is available to calculate the optimal deployment.

This paper presents an automated, flexible solution for disconnected operation that increases the availability of the system in the presence of connectivity losses. Our solution takes into account that limited information about the system and finite (usually small) amount of time to perform the redeployment task will be available. We directly leverage a software system's architecture in accomplishing this task. *Software architectures* provide abstractions for representing the structure, behavior, and key properties of a software system [20] in terms of the system's *components*, their interactions (*connectors*), and their *configurations* (*topologies*).

We increase a system's availability by (1) monitoring the system; (2) estimating the redeployment architecture; and (3) effecting that architecture. Since estimating the optimal deployment architecture is exponentially complex, we provide a fast approximative solution for increasing the system's availability, and provide an assessment of its performance. We provide a light-weight, efficient, and adaptable *architectural middleware,* called Prism-MW, that enables implementation, execution, monitoring, and automatic (re)deployment of software architectures in the Prism setting. We have evaluated our approach on a series of examples.

The remainder of the paper is organized as follows. Section 2 presents overviews of the techniques that enable our approach, and of related work.Section 3 defines the problem our work is addressing and presents an overview of our approach. Section 4 introduces Prism-MW and its foundation for the disconnected operation support. Sections 5, 6, and 7 present the three stages of the redeployment process: monitoring,

redeployment estimation, and redeployment effecting. The paper concludes with the discussion of future work.

2. Foundational and Related Work

This section presents three cases of techniques that form the foundation of our approach and a brief overview of existing disconnected operation techniques.

2.1. Deployment

Carzaniga et. al. [4] propose a comparison framework for software deployment techniques. They identify eight activities in the software deployment process, and compare existing approaches based on their coverage of these activities:

- *Release* – preparing a system for assembly and transfer to the consumer site;
- *Install* – the initial insertion of a system into a consumer site;
- *Activate* – starting up the executable components of the system at the consumer site;
- *Deactivate* – shutting down any executing components of an installed system;
- *Update* – renewing a version of a system;
- *Adapt* – modifying a software system that has previously been installed. Unlike update, adaptation is initiated by local events, such as change in the environment of the consumer site. As will be detailed in Section 7, our approach utilizes system adaptation;
- *Deinstall* – removal of the system from the consumer site; and
- *Retire* – the system is marked as obsolete, and support by the producer is withdrawn.

2.2. Mobility

Code mobility can be informally defined as the ability to dynamically change the binding between a code fragments and the location where it is executed. A detailed overview of existing code mobility techniques is given by Fuggetta et al. [9]. They describe three code mobility paradigms: (1) *remote evaluation* allows the proactive shipping of code to a remote host in order to be executed; (2) *mobile agents* are autonomous objects that carry their state and code, and proactively move across the network, and (3) *code-on-demand*, in which the client owns the resources (e.g., data) needed for the execution of a service, but lacks the functionality needed to perform the service. As detailed in Section 7, our approach leverages the remote evaluation paradigm.

Existing mobile code systems offer two forms of mobility. *Strong mobility* allows migration of both the code and the state of an execution unit to a different computational environment. *Weak mobility* allows code transfers across different environments; the code may be accompanied by some initialization data, but the execution state is not migrated. As described in Section 7, our approach utilizes strong mobility.

2.3. Dynamic Reconfigurability

Dynamic reconfigurability encompasses run-time changes to a software system's configuration via addition and removal of components, connectors, or their intercon-

nections. Oreizy et. al. [19] describe three causes of dynamic reconfigurability: (1) corrective, which is used to remove software faults, (2) perfective, used to enhance software functionality, and (3) adaptive, used to enact changes required for the software to execute in a new environment. They also identify four types of architectural reconfigurability: (1) component addition, (2) component removal, (3) component replacement, and (4) structural reconfiguration (i.e., recombining existing functionality to modify overall system behavior). As described in Section 7, our approach utilizes all four types of run-time architectural reconfigurability.

2.4. Related Approaches

We have performed an extensive survey of existing disconnected operation approaches, and provided a framework for their classification and comparison in [18]. In this section, we briefly summarize these techniques, and directly compare our approach to I5 [2], the only known approach that explicitly focuses on a system's deployment architecture and its impact on the system's availability.

The most commonly used techniques for supporting disconnected operation are:

- *Caching* – locally storing the accessed remote data in anticipation that it will be needed again [13];
- *Hoarding* – prefetching the likely needed remote data prior to disconnection [14];
- *Queueing remote interactions* – buffering remote, non-blocking requests and responses during disconnection and exchanging them upon reconnection [12];
- *Replication and replica reconciliation* – synchronizing the changes made during disconnection to different local copies of the same component [13]; and
- *Multi-modal components* – implementing separate subcomponents to be used during connection and disconnection [24].

None of these techniques change the system's deployment architecture. Instead, they strive to improve 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 [24].

I5 [2], proposes the use of the binary integer programming model (BIP) for generating an optimal deployment of a software application over a given network. This approach uses minimization of overall remote communication as the criterion for optimality, and therefore does not distinguish reliable, high-bandwidth from unreliable, low-bandwidth links between target hosts. Additionally, solving the BIP model is exponentially complex in the number of software components, and I5 does not attempt to reduce this complexity. This renders the approach applicable only to systems with very small numbers of software components and target hosts. I5 assumes that all characteristics of the software system and the target hardware environment are known *before* the system's initial deployment. Therefore, I5 is not applicable to systems whose characteristics, such as frequencies of interactions among software components, are either not known at design time, or may change during the system's execution.

3. Problem and Approach

3.1. Problem Definition

The distribution of software components onto hardware nodes (i.e., a system's soft-

ware *deployment architecture,* a concept illustrated in Figure 1) 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; this is clearly not the case with components distributed across different hosts. However, the reliability of connectivity (i.e., 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. 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.

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.

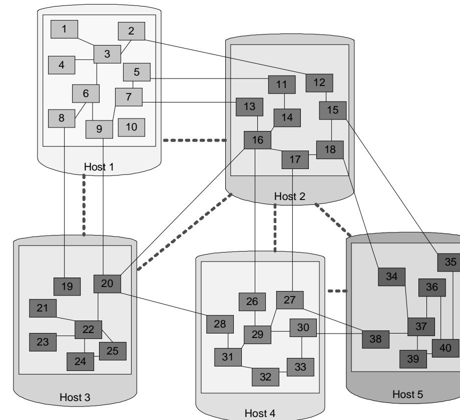

Figure 1. A sample deployment architecture with five hosts and 40 components.

In addition to hardware connectivity and frequencies of software interaction, there are other constraints on a system's redeployment, including: (1) the available memory on each host; (2) required memory for each software component; and (3) possible restrictions on component locations (e.g., a UI component may be fixed to a selected host). Figure 2 shows 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* 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. The criterion function *A* 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 two 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. Finally, 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* function.

Our approach relies on the assumption that the given system's deployment architecture is accessible from some central location. While this assumption may have been fully justified in the past, a growing number of software systems are *decentralized* to some extent. We recognize this and intend to address this problem in our future work. At the same time, before we would be able to do so, we have to understand and solve the redeployment problem in the more centralized setting.

Given:

(1) a set C of *n* components ($n = |C|$) and two functions $freq: C \times C \rightarrow R$ and $mem_{comp}: C \rightarrow R$

$$freq(c_i, c_j) = \begin{cases} 0 & \text{if } c_i = c_j \\ \text{frequency of communication between } c_i \text{ and } c_j & \text{if } c_i \neq c_j \end{cases}$$

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

(2) a set H of *k* hardware nodes ($k = |H|$) and two functions $rel: H \times H \rightarrow R$ and $mem_{host}: H \rightarrow R$

$$rel(h_i, h_j) = \begin{cases} 1 & \text{if } h_i = h_j \\ 0 & \text{if } h_i \text{ is not connected to } h_j \\ \text{reliability of the link between } h_i \text{ and } h_j & \text{if } h_i \neq h_j \end{cases}$$

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

(3) A function that restricts locations of software components $loc: C \times H \rightarrow \{0,1\}$

$$loc(c_i, h_j) = \begin{cases} 1 & \text{if } c_i \text{ can be deployed onto } h_j \\ 0 & \text{if } c_i \text{ cannot be deployed onto } h_j \end{cases}$$

Problem:

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

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

is maximized, and the following two conditions are satisfied:

(1) $\forall i \in [1, k] \qquad \left(\sum_{\substack{\forall j \in [1,n] \\ f(c_j) = h_i}} mem_{comp}(c_j) \right) \leq mem_{host}(h_i)$

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

Note that 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 both of the above two conditions.

Figure 2. Formal statement of the problem.

3.2. Our Approach

Our approach provides an automated, flexible solution for increasing the availability of a distributed system during disconnection, without the shortcomings introduced by existing approaches. For instance unlike [24], our approach does not require any recoding of the system's existing functionality; unlike [13], it does not sacrifice the correctness of computations; finally unlike [12] it does not introduce service delivery delays. We directly leverage a software system's *architecture* in accomplishing this task. We propose run-time 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. Since estimating a system's redeployment (step 2) is an exponentially complex problem, we provide an approximative solution that shows good performance.

A key insight guiding our approach is that for software systems whose frequencies of interactions among constituent components are stable over a given period of time T, and which are deployed onto a set of hardware nodes whose reliability of connectivity is also stable over T, there exists at least one deployment architecture (called the *optimal deployment architecture*) that maximizes the availability of that software system

for that target environment over the period T.

Figure 3 illustrates the system's availability during the time period *T*, in terms of our approach's three key activities. The system's initial availability is A_1. First, the system is monitored over the period T_M; during that period the availability remains unchanged. Second, during the period T_E, the system's redeployment is estimated; again system availability remains unchanged. Third, during the time period T_R the system redeployment is performed to improve its availability; the system's availability will decrease during this activity as components are migrated across hosts (and thus become temporarily unavailable). Once the redeployment is performed, the system's availability increases to A_2, and remains stable for the remainder of the time period *T*.

Performing system redeployment to improve its availability will give good results if the times required to monitor the system and complete its redeployment are negligible with respect to *T* (i.e., $T_M+T_E+T_R \ll T$). Otherwise, the system's parameters would be changing too frequently and the system would "thrash" (i.e., it would undergo continuous redeployments to improve the availability for parameter values that change either before or shortly after the redeployment is completed).

4. Prism-MW

Our approach is independent of the underlying implementation platform, but requires scalable, light-weight support for distributed architectures with arbitrary topologies. For this reason, we leverage an adaptable and extensible architecture implementation infrastructure (i.e., architectural *middleware*), called *Prism-MW* [16]. Prism-MW enables efficient and scalable implementation and execution of distributed software architectures in the Prism setting. Furthermore, Prism-MW's native support for extensibility made it highly suitable for incorporating disconnected operation support. In this section we summarize the design of Prism-MW, and describe in more detail its foundation for the disconnected operation support.

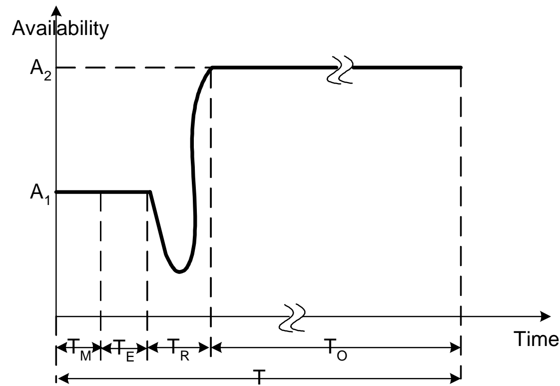

Figure 3. Graphical representation of the availability function.

4.1. Middleware Design

Prism-MW provides classes for representing each architectural element, with methods for creating, manipulating, and destroying the element. Figure 4 shows the class design view of Prism-MW. The shaded classes constitute the middleware core; the dark gray classes are relevant to the application developer. The design of the middleware is highly modular: the only dependencies among classes are via interfaces and

inheritance; the only exception is the *Architecture* class, which contains multiple *Brick*s for reasons that are explained below.

Brick is an abstract class that encapsulates common features of its subclasses (*Architecture, Component*, and *Connector*). The *Architecture* class records the configuration of its constituent components and connectors, and provides facilities for their addition, removal, and reconnection, possibly at system run-time. A distributed application is implemented as a set of interacting *Architecture* objects. *Component*s in an architecture communicate by exchanging *Event*s*,* which are routed by *Connector*s. Finally, Prism-MW associates the *IScaffold* interface with every *Brick*. Scaffolds are used to schedule events for delivery and to dispatch events using a pool of threads in a decoupled manner. *IScaffold* also directly aids architectural awareness [3] by allowing probing of the runtime behavior of a *Brick*, via different implementations of the *IMonitor* interface.

When using Prism-MW, the developer first subclasses the *Component* class for all components in the architecture and implements their application-specific methods. The next step is to instantiate the *Architecture* classes for each address space and define the needed instances of thus created components, and of connectors selected from the reusable connector library. Finally, attaching component and connector instances into a configuration is achieved by using the *weld* method of the *Architecture* class. This process can be partially automated using our tool support described in [16].

To support capabilities that may be required for different (classes of) Prism applications, Prism-MW provides three specialized classes: *ExtensibleComponent*, *ExtensibleConnector*, and *ExtensibleEvent*. These classes extend the corresponding base classes and, by composing a number of interfaces, provide the ability to select the desired functionality inside each instance of an *Extensible* class. If an interface is installed in a given *Extensible* class instance, that instance will exhibit the behavior realized inside the interface's implementation. Multiple interfaces may be installed in a single *Extensible* class instance. In that case, the instance will exhibit the combined behavior of the installed interfaces. To date, we have provided support for architectural awareness, real-time computation, distribution,

Figure 4. UML class design view of Prism-MW. Middleware core classes are highlighted. Only the relevant extensions are shown.

security, data compression, delivery guarantees, and mobility. The details of these extensions may be found in [16].

In support of distribution, Prism-MW provides the *ExtensibleConnector*, which composes the *IDistribution* interface. To date, we have provided two implementations of the *IDistribution* interface, supporting socket-based and infrared-port based inter-process communication. An *ExtensibleConnector* with the instantiated *IDistribution* interface (referred to as *DistributionConnector*) facilitates interaction across process or machine boundaries. In addition to the *IDistribution* interface inside the *Extensible-Connector* class, to support distribution and mobility we have implemented the *Serializable* interface inside each one of the *Extensible* classes. This allows us to send data as well as code across machine boundaries.

To support various aspects of architectural awareness, we have provided the *ExtensibleComponent* class, which contains a reference to *IArchitecture*. This allows an instance of *ExtensibleComponent* to access all architectural elements in its local configuration, acting as a meta-level component that effects run-time changes on the system's architecture.

4.2. Disconnected Operation Support

To date, we have augmented *ExtensibleComponent* with several interfaces. Of interest in this paper is the *IAdmin* interface used in support of redeployment. We provide two implementations of the *IAdmin* interface: *Admin*, which supports system monitoring and redeployment effecting, and *Admin*'s subclass *Deployer,* which also provides facilities for redeployment estimation. We refer to the *ExtensibleComponent* with the *Admin* implementation of the *IAdmin* interface as *AdminComponent*; analogously, we refer to the *ExtensibleComponent* with the *Deployer* implementation of the *IAdmin* interface as *DeployerComponent*.

As indicated in Figure 4, both *AdminComponent* and *DeployerComponent* contain a pointer to the *Architecture* object and are thus able to effect run-time changes to their local subsystem's architecture: instantiation, addition, removal, connection, and disconnection of components and connectors. With the help of *DistributionConnector*s, *AdminComponent* and *DeployerComponent* are able to send and receive from any device to which they are connected the events that contain application-level components (sent between address spaces using the *Serializable* interface).

In order to perform run-time redeployment of the desired architecture on a set of target hosts, we assume that a skeleton configuration is preloaded on each host. The skeleton configuration consists of Prism-MW's *Architecture* object that contains a *DistributionConnector* and an *AdminComponent* that is attached to the connector. One of the hosts contains the *DeployerComponent* (instead of the *AdminComponent*) and controls the redeployment process. The *DeployerComponent* gathers all the monitoring data from different *AdminComponents* and estimates the system redeployment. Then, the *DeployerComponent* collaborates with *AdminComponents* to effect the estimated redeployment architecture. The details of this process are described in Sections 5, 6, and 7.

Our current implementation assumes that the host containing the *DeployerComponent* will have a direct (possibly unreliable) connection with all the remaining hosts. An alternative design would require only the existence of a path between any two hosts (i.e, a connected graph of hosts). In this scenario, the information that needs to be

exchanged between a pair of hosts not connected directly would need to get routed through a set of intermediary hosts (e.g, by using event forwarding mechanisms of Siena [5]). We are currently implementing and evaluating this design.

5. System Monitoring

5.1. Monitoring Requirements

System monitoring [10] is a process of gathering data of interest from the running application. In the context of system redeployment, the following data needs to be obtained: (1) frequency of interaction among software components; (2) each components' maximum memory requirements; (3) reliability of connectivity among hardware hosts; and (4) available memory on each host. Since we assume that the available memory on each host and maximum required memory for each software component are stable throughout the system's execution, these parameters can be obtained either from the system's specification (e.g., [2]) or at the time the initial deployment of the system is performed. Therefore, the active monitoring support should gather the following parameters: (1) for each pair of software components in the system, the number of times these components interact is recorded, and (2) for each pair of hardware hosts, the ratio of the number of successfully completed remote interactions to the total number of attempted interactions is recorded. Furthermore, due to the limited time available to perform a system's redeployment, the time required to complete system monitoring should be minimized (recall Figure 3).

5.2. Prism-MW's Support for Monitoring

In support of monitoring Prism-MW provides the *IMonitor* interface associated through the *Scaffold* class with every *Brick*. This allows us to monitor the run-time behavior of each *Brick*. To date, we have provided two implementations of the *IMonitor* interface: *EvtFrequencyMonitor* records the frequencies of different events the associated *Brick* sends, while *DisconnectionRateMonitor* records the reliability of connectivity between its associated *DistributionConnector* and other remote *DistributionConnectors*.

An *AdminComponent* on any device is capable of accessing the monitoring data of its local components and connectors (recorded in their associated implementations of the *IMonitor* interface) via its pointer to the *Architecture*. The *AdminComponent* then sends that data in the form of serialized *ExtensibleEvent*s to the requesting *DeployerComponent*.

In order to minimize the time required to monitor the system, system monitoring is performed in short intervals. The *AdminComponent* compares the results from consecutive intervals. As soon as the difference in the monitoring data between two consecutive intervals becomes small (i.e., less than a given value ε), the *AdminComponent* assumes that the monitoring data is stable, and informs the *DeployerComponent*.

Note that the *DeployerComponent* will get the reliability data twice (once from each host). On the one hand, this presents additional overhead. On the other hand, this feature can be used to more accurately assess the reliability data, by taking the average from the two sets of monitoring data. Furthermore, this overhead presents only a frac-

tion of the total monitoring overhead, since the number of hosts is usually much smaller than the number of components. The frequency data will be received by the *DeployerComponent* only once, since each *EvtFrequencyMonitor* only monitors the outgoing events of its associated component.

An issue we have considered deals with cases when most, but not all system parameters are stable. As described above, if any of the parameters do not satisfy their ε constraint, the redeployment estimation will not be initiated. There are at least two ways of addressing this situation. First is to increase the ε for the specific troublesome parameters and thus induce the redeployment estimation. Alternatively, a single, global ε_g may be used to initiate redeployment estimation as soon as the average difference of the monitoring data for all the parameters in the system becomes smaller than ε_g. We support both these options and are currently assessing their respective strengths and weaknesses.

Our initial assessment of Prism-MW's monitoring support suggests that *continuous* monitoring on each host will likely induce no more than 10% computational overhead and 5% memory overhead. We are currently studying the *actual* monitoring overhead caused by our solution.

6. Estimating Redeployment

6.1. Algorithms and Their Analysis

In this section we describe our two algorithms for estimating a system's redeployment architecture. These algorithms require the data obtained during the monitoring stage. We analyze the algorithms' performance, both in terms of time complexity (i.e., T_E) and of the achieved availability.

6.1.1. Exact Algorithm

This algorithm tries every possible deployment architecture, and selects the one that has maximum availability and satisfies the constraints posed by the memory and restrictions on locations of software components. The 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, however, this algorithm may be computationally too expensive if the number of hardware nodes and unfixed software components involved is not very small. Figure 5 shows the

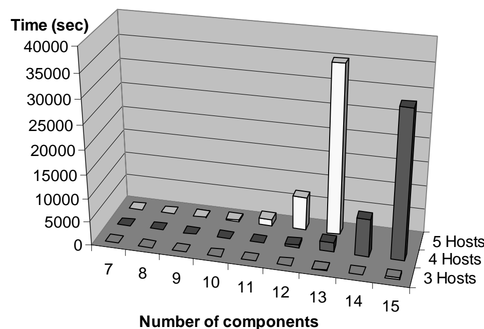

Figure 5. Performance benchmark of the exact algorithm on an Intel Pentium III, 850MHz, 128 MB of RAM running Java JDK 1.1.8 on Windows XP.

performance of this algorithm. For example, if there are no restrictions on locations of software components, even for a relatively small deployment architecture (15 components, 4 hosts), the exact algorithm runs for more than eight hours.

This algorithm may be used for calculating optimal deployment for systems whose characteristics (i.e., input parameters to the algorithm) are stable for a very long time period. In such cases, it may be beneficial to invest the time required for the exact algorithm, in order to gain maximum possible availability. However, note that even in such cases, running the algorithm may become infeasible very quickly.

6.1.2. Approximative Algorithm

Given the complexity of the exact algorithm and limited time available for estimating the system's redeployment, we had to devise an approximative algorithm that would significantly reduce this complexity while exhibiting good performance.

The redeployment problem is an instance of a large class of global optimization problems, in which the goal is to find a solution that corresponds to the minimum (or maximum) of a suitable criterion function, while it satisfies a given collection of feasibility constraints. Most global optimization problems are NP-hard [7].[1]

Different techniques have been devised to search for approximative solutions to global optimization problems. Some of the more commonly used strategies are dynamic programming, branch-and-bound, and greedy algorithms [6]. When devising approximative solutions for global optimization problems, the challenge is to avoid getting stuck at the "local optima". There are several techniques that can be applied to avoid this problem: genetic algorithms, simulated annealing, and stochastic (i.e., random) algorithms [6,21]. It has been demonstrated that stochastic algorithms produce good results more quickly than the other two techniques [1]. In this section, therefore, we describe and assess the performance of a stochastic approximative algorithm with polynomial time complexity $(O(n^2))$.

This algorithm randomly orders all the hosts and randomly orders all the components. Then, going in order, it assigns as many components to a given host as can fit on that host, ensuring that the assignment of each component is allowed (recall the *loc* restriction in Figure 2). 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 is repeated a desired number of times, and the best obtained deployment is selected. The complexity of this algorithm is polynomial, since we need to calculate the availability for every deployment, and that takes $O(n^2)$ time.

In order to assess the performance of our two algorithms, we have implemented a tool, called DeSi [16], that provides random generation of the system parameters, the ability to modify these parameters manually, and the ability to both textually and graphically display the results of the two algorithms.

We have assessed the performance of the approximative algorithm by comparing it against the exact algorithm, 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 problems, the approximative algorithm invariably found a solution that was at least 80% of the optimal with 1000 iterations. Figure 6 shows the results of these benchmarks.

For larger problems, where the exact algorithm is infeasible, we have compared the

---

[1] An NP-hard problem cannot be solved, or an existing solution to it verified, in polynomial time.

results of the approximative algorithm for varying number of iterations to the minimum availability obtained.[2] The results of this benchmark are illustrated in Figure 7. The achieved availability in these four architectures was at least 70%. Furthermore, with an order of magnitude increase in the number of iterations, the output of the algorithm improved at most by 2%. Finally, the achieved availability was at least 60% greater than the minimum.

6.2. Prism-MW's support for estimation

The *DeployerComponent* accesses its local monitoring data; it also receives all the monitoring data from the remote *AdminComponents*. Once the monitoring data is gathered from all the hosts, the *DeployerComponent* initiates the redeployment estimation, by invoking the *execute* operation of the installed *IDeployerAlgorithm* interface. To date, we have provided two implementations of the *IDeployerAlgorithm* interface, *ExactAlgorithm* and *ApproximativeAlgorithm*. The output of each of these algorithms is a desired deployment architecture (in the form of unique component-host identifier pairs), which now needs to be effected.

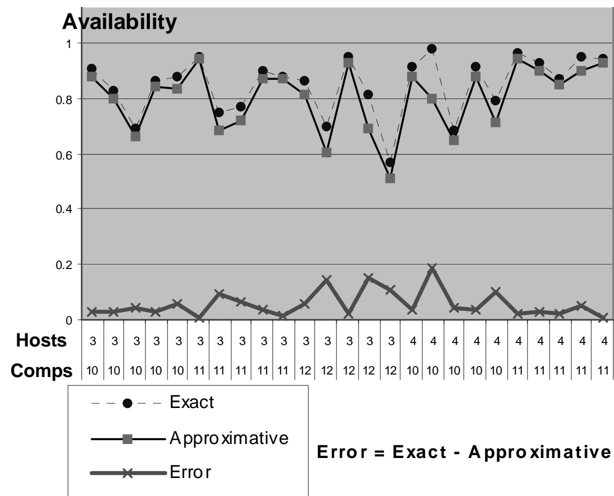

Figure 6. Comparing the performance of the exact and approximative algorithms for randomly generated architectures with three or four hosts and ten, eleven, or twelve components

7. Effecting Redeployment

7.1. Requirements for Effecting Redeployment

Effecting the system's redeployment is performed by determining the difference between the current deployment architecture and the estimated one. This will result in a set of components to be migrated. Thus obtained components are then migrated from their source hosts to the destination hosts. After the migration is performed, the migrant components need to be attached to the appropriate locations in their destination configurations. In order to effectively manage system redeployment, the exchange of components between hosts that are not directly connected should be supported. Furthermore, a solution for effecting redeployment would also need to address situations

[2] The minimum availability is obtained by recording the worst availability during the approximative algorithm's search.

in which network bandwidth and reliability of connectivity between a pair of hosts restrict the maximum size of components that may be exchanged. Otherwise, it may not be possible to effect the redeployment architecture obtained during the estimation phase.

7.2. Prism-MW's Support for Redeployment Effecting

The *DeployerComponent* controls the process of effecting the redeployment as follows:

1. The *DeployerComponent* sends events to inform *AdminComponent*s of their new local configurations, and of remote locations of software components required for performing changes to each *AdminComponent*'s local configuration.
2. Each *AdminComponent* determines the difference between its current configuration and the new configuration, and issues a series of events to remote *AdminComponent*s requesting the set of components that are to be deployed locally. If some of the devices containing the desired components are not directly reachable from the requesting device, the request events for those components are sent by the local *AdminComponent*s to the *DeployerComponent*. The *DeployerComponent* then forwards those events to the appropriate destinations, and forwards the responses containing migrant components to the requesting *AdminComponents*. Therefore, the *DeployerComponent* serves as a router for devices not directly connected (recall the discussion in Section 4.2).
3. Each *AdminComponent* that receives an event requesting its local component(s) to be deployed remotely, detaches the required component(s) from its local configuration, serializes them (therefore preserving the component's state during the migration), and sends them as a series of events via its local *DistributionConnector* to the requesting device.
4. The recipient *AdminComponent*s reconstitute the migrant components from the received events.
5. Each *AdminComponent* invokes the appropriate methods on its *Architecture* object to attach the received components to the local configuration.

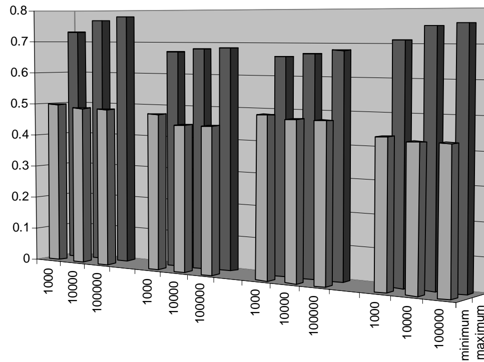

Figure 7. Comparing the performance of the approximative algorithm for four randomly generated architectures with 10 hosts and 100 components, with 1000, 10,000, and 100,000 iterations.

As discussed in Section 4.2, our current implementation assumes a centralized organization, i.e., that the device containing the *DeployerComponent* will have direct connection with all the remaining devices. We are currently implementing and evaluating an existing decentralized solution to a similar problem [5].

To address the situations where the reliability of connectivity, network bandwidth, and component size would prevent the exchange of a migrant component between a pair of hosts from occurring atomically (i.e., using a single event to send the migrant component), Prism-MW supports incremental component migration, as follows:

1. The sending *AdminComponent* serializes the migrant component into a byte stream.
2. The sending *AdminComponent* divides the byte stream into small segments, whose size is programmatically adjustable.
3. Each segment is packaged into a separate event, numbered, and sent atomically.
4. After the last chunk is sent, the sending *AdminComponent* sends a special event denoting that the entire component has been sent.
5. The receiving *AdminComponent* reconstitutes the migrant component from the received byte code chunks, requesting that the sending *AdminComponent* resend any missing (numbered) segments.

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. On the one hand, a number of promising solutions to the disconnected operation problem have already emerged. On the other hand, these solutions have focused on specific system aspects (e.g., *data* caching, hoarding, and replication; special purpose, disconnection-aware *code*) and operational scenarios (e.g., anticipated disconnection [24]). While each of these solutions may play a role in the emerging world of highly distributed, mobile, resource constrained environments, 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 its *deployment architecture*.

This paper has thus presented a set of algorithms, techniques, and tools for improving a distributed, mobile system's availability via redeployment. Our support for disconnected operation has been successfully tested on several example applications. We are currently developing a simulation framework, hosted on Prism-MW, that will enable the assessment of our approach on a large number of simulated hardware hosts, with varying but controllable connectivity among them.

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) devising new approximative algorithms targeted at different types of problems (e.g., different hardware configurations such as star, ring, and grid), (2) support for decentralized redeployment (in cases where the *DeployerComponent* is not connected to all the remaining hardware hosts), (3) addressing the issue of trust in performing distributed redeployment, and (4) studying the trade-offs between the cost of redeployment and the resulting improvement in system availability. Finally, we intend to expand our solutions to include system parameters other than memory, frequency of interaction, and reliability of connection (e.g., battery power, display size, system software available on a given host, and so on).

9. Acknowledgements

The authors wish to thank Sam Malek for his contributions to the Prism project. This material is based upon work supported by the National Science Foundation under Grant Numbers CCR-9985441and ITR-0312780.

References

[1] J.T. Alander. Empirical comparison of stochastic algorithms. *Second Nordic Workshop on Genetic Algorithms and their Applications*. August 1996, Vaasa, Finland.

[2] M. C. Bastarrica, et.al. A Binary Integer Programming Model for Optimal Object Distribution. *Second International Conference on Principles of Distributed Systems*, Amiens, France, December 1998.

[3] L. Capra, W. Emmerich and C. Mascolo. Middleware for Mobile Computing. *UCL Research Note* RN/30/01.

[4] A. Carzaniga et. al. A Characterization Framework for Software Deployment Technologies. *Technical Report*, Dept. of Computer Science, University of Colorado, 1998.

[5] A. Carzaniga and A. L. Wolf. Forwarding in a Content-Based Network. *SIGCOMM '03*. Karlsruhe, Germany. August 2003.

[6] T. H. Cormen, C. L. Leiserson and R. L. Rivest. Introduction to Algorithms. *MIT Press*, Cambridge, MA, 1990.

[7] P. Crescenzi et al. A Compendium of NP Optimization Problems. http://www.nada.kth.se/~viggo/problemlist/

[8] F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. I*EEE Transactions on Software Engineering*, June 1976.

[9] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, May 1998.

[10] D. Garlan, et. al. Using Gauges for Architecture-Based Monitoring and Adaptation. *Working Conf. on Complex and Dynamic Systems Arch.,* Brisbane, Australia, December 2001.

[11] Hyperdictionary. http://hyperdictionary.com/dictionary/

[12] A. D. Joseph, et. al., Rover: a toolkit for mobile information access, *15th ACM Symposium on Operating Systems Principles*, December 1995, Colorado.

[13] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, vol. 10, no. 1, February 1992.

[14] G. H. Kuenning and G. J. Popek. Automated Hoarding for Mobile Computers. Proceedings of the *16th ACM Symposium on Operating Systems Principles,* St. Malo, France, October, 1997.

[15] E. A. Lee. Embedded Software. In M. Zelkowitz, Ed., *Advances in Computers,* Vol. 56, Academic Press, London, 2002.

[16] 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.

[17] M. Mikic-Rakic and N. Medvidovic. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. *Middleware 2003*. Rio De Janeiro, Brazil, June 2003.

[18] M. Mikic-Rakic and N. Medvidovic. Toward a Framework for Classifying Disconnected Operation Techniques. *ICSE Workshop on Architecting Dependable Systems*, Portland, Oregon, May 2003.

[19] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based run-time Software Evolution. *20th Int. Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.

[20] D.E. Perry, and A.L. Wolf. Foundations for the Study of Software Architectures. *Software Engineering Notes*, Oct. 1992.

[21] S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. *Prentice Hall,* Englewood Cliffs, NJ, 1995.

[22] H. Saran, and V. Vazirani. Finding k-cuts Within Twice the Optimal. *32nd Annual IEEE Symposium on Foundations of Computer Science*, 743-751, 1991.

[23] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice-Hall*, 1996.

[24] Y. Weinsberg, and I. Ben-Shaul. A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices. *Int. Conf. on Software Engr.*, Orlando, Florida, May 2002.

[25] 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.