# Support for Disconnected Operation via Architectural Self-Reconfiguration

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 state of the target hardware environment and may need to be altered to improve the software system's availability. In this paper, we present a flexible, software architecture-based solution for disconnected operation that increases the availability of a system during disconnection by allowing the system to monitor and automatically redeploy itself.*

## 1. Introduction

The emergence of mobile devices, such as portable computers, hand-held PDAs, and mobile phones, and various wireless networking solutions make distributed computation possible anywhere. Applications involving these devices are highly dependent on the underlying network. Unfortunately, network connectivity failures are not rare: mobile devices face frequent and unpredictable connectivity losses due to their constant location change and lack of network coverage; the costs of wireless connectivity often also induce voluntary, user-initiated disconnection; and even the highly reliable WAN and LAN connectivity is unavailable 1.5% to 3.3% of the time [16].

For this reason, highly distributed and mobile systems are challenged by the problem of *disconnected operation* [15], where the system must continue functioning in the temporary absence of network connectivity. Disconnected operation forces subsystems executing on each network host to temporarily operate independently from other (disconnected) hosts. This presents a major challenge for software systems that are highly dependent on network connectivity because the 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 [6]. In the context of highly distributed, mobile environments, where a most common cause of (partial) system inaccessibility is 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 over a period of time.

In this context, a key observation is that the distribution of software components onto hardware nodes (i.e., a system's software *deployment architecture*) greatly influences the system's availability in the face of connectivity losses. For example, in such cases it is desirable to collocate components that interact frequently. However, the parameters that influence the optimal distribution of a system (e.g., the reliability of network links) may change during the system's execution. For this reason, the existing software deployment architecture may be ill-suited for the given target hardware environment, and the system may need to be *redeployed* to improve its availability. However, determining a software system's deployment that will maximize its availability in the face of disconnection (i.e., the *optimal* deployment) is an exponentially complex problem: in the most general case its complexity is $k^n$, where $k$ is the number of hardware hosts and $n$ the number of software components.

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 two observations: (1) system parameters that influence the optimal deployment architecture may change significantly during the system's execution, and (2) finite (usually small) amount of time will be available to perform the redeployment task once the change in system parameters occurs. 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 [14] in terms of the system's *components*, their interactions (*connectors*), and their *configurations* (*topologies*).

Our solution allows a system to increase its availability autonomously, by actively (1) monitoring itself, (2) estimating its new deployment architecture, and (3) effecting that architecture. To this end, we adapt a light-weight, efficient, and tailorable *architectural middleware,* called Prism-MW [12], that enables implementation, execution, monitoring, and (re)deployment of software architectures in highly distributed, mobile settings. We have evaluated our approach on a series of benchmark examples.

The remainder of the paper is organized as follows. Section 2 presents an overview 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 the system, estimating the redeployment, and effecting the redeployment. The paper concludes with a discussion of future work.

## 2. Related Work

We have performed an extensive survey of existing disconnected operation approaches, and provided a framework for their classification and comparison [13]. In this section, we briefly summarize these techniques, and directly compare our approach to two techniques that explicitly focus on a system's deployment architecture and its impact on 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 [8];
- *Hoarding* – prefetching the likely needed remote data prior to disconnection [9];
- *Queueing remote interactions* – buffering remote, non-blocking requests and responses during disconnection and exchanging them upon reconnection [7];
- *Replication and replica reconciliation* – synchronizing the changes made during disconnection to different local copies of the same component [8]; and
- *Multi-modal components* – implementing separate subcomponents to be used during disconnection [15].

None of these techniques changes the system's deployment architecture. Instead, they strive to improve the system's availability by sacrificing either correctness (in the case of replication) or service delivery time (queueing), or by requiring implementation-level changes to the existing application's code [15].

I5 [1] proposes the use of the binary integer programming model (BIP) for generating an optimal deployment of a software application over a given network. I5 does not distinguish reliable, high-bandwidth from unreliable, low-bandwidth links between hosts. Additionally, solving the BIP model is exponentially complex in the number of software components, and I5 does not attempt to reduce this complexity. I5 assumes that all characteristics of the software system and the hardware environment are known *before* the system's initial deployment. Therefore, I5 is not applicable to systems whose characteristics are either not known at design time, or may change during the system's execution. Finally, I5 does not provide facilities for effecting or modifying the deployment architecture.

Haas et. al. [5] provide a scalable framework for autonomic service deployment in networks. Hardware nodes are described in terms of properties (e.g., bandwidth, CPU speed). The authors propose a three step process for autonomic deployment: (1) network distribution of requirements for a new service, (2) summarization of how each node can contribute to supporting that service and selection of most appropriate hosts, and (3) installation of a service on the most appropriate hosts. This approach does not address the inherent exponential complexity in the selection of the most appropriate deployment, or that properties of services and hosts 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 software *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. However, the reliability of connectivity among the "target" hardware nodes on which the system is deployed (modeled in terms of the rate of network failure) is usually not known before the deployment and may change during the system's execution. The frequencies of interaction among software components may also be unknown. For this reason, the currently executing software deployment architecture may be ill-suited for the given state of the 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 for the given target environment (referred to as *optimal deployment architecture*) is an exponentially complex problem.

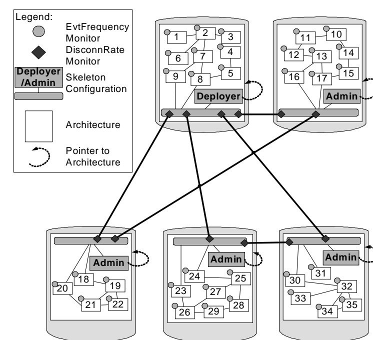


**Figure 1.** A sample deployment architecture with five hosts and 35 components.

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 (co)locations (e.g., two components may not be allowed to reside on the same host). Figure 2 shows a formal definition of the problem. An in-depth discussion of this problem, its generalization, and a detailed overview of possible approximation techniques is given in [11].

The described redeployment problem assumes that the main source of degradation of a system's availability is disconnection, i.e., that the system's availability would not vary significantly between different deployments across a completely reliable network. This assumption implies that the network bandwidth and volume of exchanged data between components will not influence the system's availability. We can still address situations in which this assumption does not hold via the *loc* and *colloc* functions if the number of components exchanging high volumes of data is reasonably small. Using these functions, valid deployments

| Model | Definition |
|---|---|
| Given:<br>(1) a set $C$ of $n$ components ($n=\lvert C\rvert$) and two functions $freq: C\times C\to\Re$ and $mem_{comp}: C\to\Re$<br>$freq(c_i,c_j)=\left\{\begin{array}{ll} 0 & if \quad c_i=c_j \\ frequency\ of\ communication\ between\ c_i\ and\ c_j & if \quad c_i\neq c_j\end{array}\right\}$<br>$mem_{comp}(c)=required\ memory\ for\ c$<br><br>(2) a set H of $k$ hardware nodes ($k=\lvert H\rvert$) and two functions $rel: H\times H\to\Re$ and $mem_{host}: H\to\Re$<br>$rel(h_i,h_j)=\left\{\begin{array}{l} 1 \quad if \quad h_i=h_j \\ 0 \quad if \quad h_i\ is\ not\ connected\ to\ h_j \\ reliability\ of\ the\ link\ between\ h_i\ and\ h_j \quad if \quad h_i\neq h_j\end{array}\right\}$<br>$mem_{host}(h)=available\ memory\ on\ host\ h$<br><br>(3) Two functions that restrict locations of software components<br>$loc: C\times H\to\{0,1\}\quad colloc: C\times C\to\{-1,\ 0,\ 1\}$<br>$loc(c_i,h_j)=\left\{\begin{array}{l} 1 \quad if \quad c_i\ can\ be\ deployed\ onto\ h_j \\ 0 \quad if \quad c_i\ cannot\ be\ deployed\ onto\ h_j\end{array}\right\}$<br>$colloc(c_i,c_j)=\left\{\begin{array}{l} -1 \quad if \quad c_i\ cannot\ be\ on\ the\ same\ host\ as\ c_j \\ 1 \quad if \quad c_i\ has\ to\ be\ on\ the\ same\ host\ as\ c_j \\ 0 \quad if \quad there\ are\ no\ restrictions\ on\ collocation\ of\ c_i\ and\ c_j\end{array}\right\}$ | Problem:<br>Find a function $f: C\to H$ such that the system's overall availability $A$ defined as<br>$A=\dfrac{\sum_{i=1}^{n}\sum_{j=1}^{n}(freq(c_i,c_j)*rel(f(c_i),f(c_j)))}{\sum_{i=1}^{n}\sum_{j=1}^{n}freq(c_i,c_j)}$<br>is maximized, and the following three conditions are satisfied:<br>(1) $\forall i\in[1,k]\left\{\forall j\in[1,n]\ \ f(c_j)=h_i \ \middle\vert\ \sum_j mem_{comp}(c_j))\le mem_{host}(h_i)\right\}$<br><br>(2) $\forall j\in[1,n] \qquad loc(c_j,f(c_j))=1$<br><br>(3) $\forall k\in[1,n] \quad \forall l\in[1,n]$<br>$if \quad (colloc(c_k,c_l)=1)\Rightarrow(f(c_k)=f(c_l))$<br>$if \quad (colloc(c_k,c_l)=-1)\Rightarrow(f(c_k)\neq f(c_l))$<br><br>In the most general case, the number of possible functions *f* is $k^n$. However, note that some of these deployments may not satisfy one or more of the above three conditions. |

**Figure 2.** Formal statement of the problem.

can be restricted such that components that exchange high volumes of data reside either on the same host, or on hosts between which the network bandwidth is sufficient.

Our approach also 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 are addressing this problem in our on-going work. At the same time, before doing so, we have had to understand and solve the redeployment problem in the more centralized setting.

## 3.2. Our Approach

We provide a solution for increasing the availability of a distributed system during disconnection, without the shortcomings of the existing approaches. For instance, unlike [15] our approach does not require any recoding of the system's existing functionality or human intervention; unlike [8] it does not sacrifice the correctness of computations; finally, in comparison to [7] it minimizes service delivery delays. We directly leverage a software system's *architecture* in accomplishing this task. We employ autonomous, run-time redeployment to increase a system's availability by enabling the system to (1) monitor its operation, (2) estimate its new deployment architecture, and (3) effect the estimated architecture. Since estimating a system's optimal deployment (step 2) is an exponentially complex problem, we provide a set of approximative algorithms with different levels of trade-off between complexity and achieved availability. We support automated selection of the most appropriate algorithm for a given redeployment situation.

A key insight guiding our approach is that, for software systems whose frequencies of interaction 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 optimal deployment architecture for the period T. Figure 3 illustrates a 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 monitors its operation over the period $T_M$; during that period the availability remains unchanged. Second, during the period $T_E$, the system estimates its redeployment; again, availability remains unchanged. Third, during the time period $T_R$ the system's 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.* Once the system parameters change the same process repeats (illustrated in Figure 3 with time period T').

Performing system redeployment to improve its availability will give good results if the times required to moni-

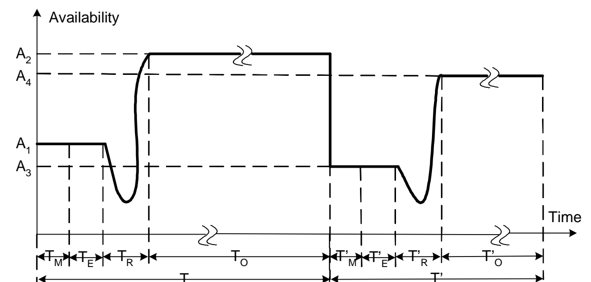


**Figure 3.** Graphical representation of the availability function.

tor the system and complete its redeployment are small with respect to $T$ (i.e., $T_M+T_E+T_R << T$). Otherwise, the system's parameters would be changing too frequently and the system would undergo continuous redeployments to improve the availability for parameter values that change either before or shortly after the redeployment is completed. In such cases, another technique (e.g., caching or hoarding) might be more appropriate. However, we suspect that availability will remain a problem in such highly unstable systems regardless of the employed technique(s).

## 4. Prism-MW

Our approach is independent of the underlying implementation platform, but requires scalable support for distributed software architectures. For this reason, we leverage an extensible architecture implementation infrastructure (i.e., architectural *middleware*), called *Prism-MW* [12], which enables efficient implementation, deployment, and execution of distributed software architectures. In this section we summarize the design of Prism-MW, and describe its foundation for the disconnected operation support.

### 4.1. Middleware Design

Prism-MW has been designed as an object-oriented framework that provides classes for representing each architectural element. Figure 4 shows the class design view of Prism-MW. *Brick* is an architecture-level system building block. It is implemented as 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 and dispatch events using a pool of threads in a decoupled manner. *IScaffold* also directly aids architectural self-awareness by allowing the runtime behavior of a *Brick* to be probed, via different implementations of the *IMonitor* interface.

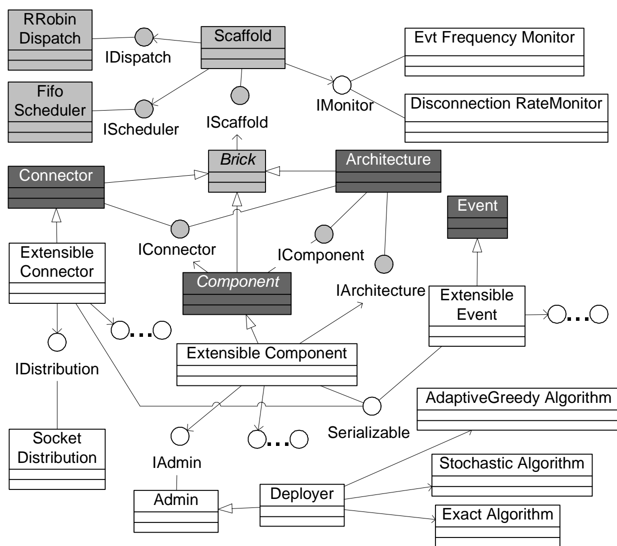


**Figure 4.** UML class design view of Prism-MW. Middleware core classes are shaded. The four dark gray classes are used by the application developer. Only the relevant extensions are shown.

To support capabilities that may be required for distributed and mobile 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. To date, we have provided support for self-awareness, real-time computation, distribution, security, data compression, delivery guarantees, and mobility [12].

In support of distribution, Prism-MW provides the *ExtensibleConnector*, which composes the *IDistribution* interface. An *ExtensibleConnector* with the instantiated *IDistribution* interface (referred to as *DistributionConnector*) facilitates interaction across process or machine boundaries. In addition, to support distribution and mobility we have implemented the *Serializable* interface inside each one of the three *Extensible* classes. This allows us to send data, as well as code and system state across machine boundaries.

To support various aspects of architectural self-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 can automatically effect 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 reference to *IArchitecture* 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* attached to the connector (recall Figure 1). One of the hosts contains the *DeployerComponent* (instead of the *AdminComponent*), which maintains a model of the system's current deployment architecture and controls the redeployment process. The *DeployerComponent* receives the results of monitoring from different *AdminComponents* and estimates the system new deployment. Then, the *DeployerComponent* collaborates with *AdminComponents* to effect the estimated deployment architecture. The details of this process are described in Sections 5, 6, and 7. In our Java implementation of Prism-MW, the overhead of each *AdminComponent* is 9 KB, while each *DeployerComponent* requires 10 KB of memory. The overhead of a single *DistributionConnector* with a single remote connection is 13.5 KB. Therefore, the total overhead of the skeleton configuration is on average 23 KB.

Our current implementation assumes that the host containing the *DeployerComponent* will have direct (possibly unreliable) connections 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 [2]). We are currently implementing and evaluating this design.

## 5. System Monitoring

Prism-MW provides the *IMonitor* interface associated through the *Scaffold* class with every *Brick*. This allows for autonomous, active monitoring of a *Brick*'s run-time behavior. 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 *DistributionConnector*s using a common "pinging" technique (e.g., [3]).

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 reference to *IArchitecture*. The *AdminComponent* then sends that data in the form of serialized *ExtensibleEvent*s to the *DeployerComponent*. In order to minimize the time required to monitor the system, monitoring is performed in short intervals of adjustable duration. The *AdminComponent* actively compares the results from consecutive intervals. As soon as the difference in the monitoring data between a desired number of consecutive intervals becomes small (i.e., less than an adjustable value $\varepsilon$), the *AdminComponent* assumes that the monitoring data is stable, and informs the *DeployerComponent*.

Note that the *DeployerComponent* will get the network reliability data twice (once from each of two connected hosts). On the one hand, this presents additional overhead. On the other hand, this feature can be used to more accurately assess the reliability data, e.g., by taking the average from the two sets of monitoring data. Furthermore, this overhead presents only a fraction 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 fail to satisfy their $\varepsilon$ constraint, the redeployment estimation will not be initiated. There are at least two ways of addressing this situation. The first is to increase the $\varepsilon$ for the "troublesome" parameters and thus induce the redeployment estimation. Alternatively, a single, global $\varepsilon_g$ may be used to initiate the estimation as soon as the average difference of the monitoring data for all the parameters in the system becomes smaller than $\varepsilon_g$. We support both these options and are currently assessing their respective strengths and weaknesses.

Our assessment of Prism-MW's monitoring support suggests that *continuous* monitoring on each host will induce less than 10% computational overhead and 5% memory overhead on a system. The actual monitoring overhead caused by our solution depends on the duration and frequency of monitoring intervals, and can be negligible (as little as 0.1%) for systems whose rate of change in the monitored parameters is reasonably uniform.

## 6. Estimating Redeployment

### 6.1. Algorithms for Redeployment Estimation

In this section we briefly describe several algorithms we have developed for estimating a system's new deployment architecture. These algorithms require the data obtained during the monitoring stage. A detailed performance comparison of these algorithms is given in [11].

**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, this algorithm may be computationally too expensive unless the number of hardware nodes and unfixed software components is very small. For example, even for a relatively small

deployment architecture (15 components, 4 hosts), a Java JDK 1.4 implementation of the exact algorithm runs for more than eight hours on a mid-range PC.

**Stochastic Algorithm.** 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* and *colloc* restrictions 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.

**Adaptive 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. This is achieved by selecting the "best" host and "best" component at each step. 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 for assignment to a host is performed by choosing the component with the highest frequency of interaction with other components in the system, and the lowest required memory. Our algorithm automatically assigns different calibration factors to each one of these parameters, as detailed in [11]. The calibration factors denote the degree of "contribution" of each parameter to the selection of the best host and component.

Once found, the best component is assigned to the best host, making certain that the (co-)location constraints are satisfied. The algorithm proceeds with searching for the next best component among the remaining, unassigned components, until the best host is full. Next, the algorithm selects the best host among the remaining, unassigned hosts. This process repeats until every component has been assigned to a host. The complexity of the approximative algorithm in the most general case (i.e., when no components are fixed to a single host, and there are more components than hosts) is $O(n^3)$ [11].

**Performance analysis.** To assess the performance of these algorithms, we have implemented a tool that supports: (1) random generation of the input parameters for the algorithms; (2) modification of the generated input parameters; (3) specification of different constraints on software component locations; (4) invocation of the exact, stochastic, or adaptive greedy algorithms; and (5) automated execution of benchmarks to asses and compare the performance of the algorithms [11].

We have used this tool to evaluate the algorithms on a number of different, randomly generated architectures. As

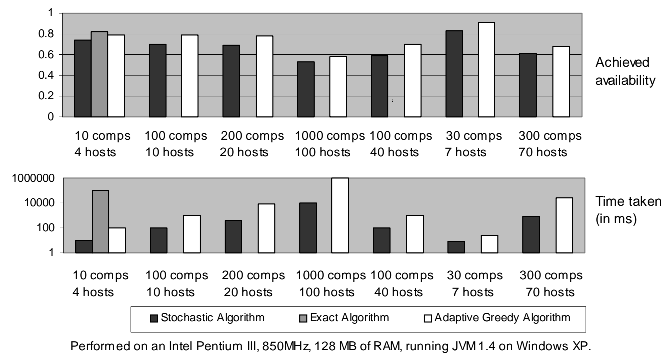


**Figure 5.** Performance analysis of different algorithms for seven different, randomly created architectures.

illustrated in Figure 5, the described algorithms exhibit different performance both in terms of produced availability and execution time. The slowest, exact algorithm finds the optimal availability (but can take *years* to do so for any but very small systems, hence its results are only shown for the smallest architecture in Figure 5); the adaptive greedy algorithm is slower than the stochastic (by the factor of $n$), but it also achieves higher availability. These observations are leveraged by the *DeployerComponent* in deciding which algorithm to use in a given situation, as described below.

## 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 *AdminComponent*s. Once the monitoring data is gathered from all the hosts, the *DeployerComponent* initiates the redeployment estimation by invoking the *execute* operation of one of the three algorithms (*Exact*, *Stochastic*, or *AdaptiveGreedy*, shown in Figure 4). The *DeployerComponent* decides on the algorithm to invoke based on the following information:

(1) $T_{AVG}$, the average duration of time period T – this value is obtained by observing the history of T's duration over the system's execution;

(2) $A_C$, the current availability – this value is calculated by the *DeployerComponent* from the monitoring data using the formula shown in Figure 2;

(3) $A_{exp}$, the expected improvement in availability achieved by the different algorithms (i.e., $A_E$, $A_S$, $A_G$, see Figure 6) – these values have been obtained empirically (see Figure 5) and are provided to the *DeployerComponent* at its start-up time; and

(4) $T_E$, the running time of each algorithm (i.e., $T_{EE}$, $T_{ES}$, $T_{EG}$, see Figure 6) – these values are also available from the benchmarking data (see Figure 5); from this data they can be estimated for an arbitrary system, based on each algorithm's computational complexity.

The algorithm to be invoked is the one whose overall resulting availability over the expected remainder of the time period T is the greatest, i.e., whose value of

$$T_E * A_C + (T_{AVG} - T_E) * A_{exp}$$

is greatest. The surface area corresponding to this value for the stochastic algorithm is shaded in Figure 6.

Note that the above formula does not take into account the area corresponding to the time period $T_R$. The reason for this is two-fold. First, the magnitude of the "dip" in availability is proportional to the number and sizes of components that need to be migrated in order to effect the new deployment (see Section 7). However, calculating that value would require running all three algorithms, which is precisely what we are trying to avoid in performing our estimation. Secondly, we have empirically demonstrated that the values of $T_R$ produced by the three algorithms are comparable on average.

Also note that immediately after a system's initial deployment we will not know its $T_{AVG}$. Therefore, in our implementation either the adaptive greedy algorithm or the stochastic algorithm is randomly selected initially. Note also that our solution does not restrict the number or kinds of algorithms one can employ for redeployment estimation. Finally, note that the selection of the algorithm will depend on the length of time period *T*.

The above strategy allows the DeployerComponent to quickly calculate the "winning" algorithm. At the same time, some of the parameters used in the process (e.g.,$A_{exp}$) are very coarse approximations and may suggest a suboptimal choice of the algorithm. There are a number of other strategies one can implement to support automated invocation of the most appropriate algorithm. We are currently evaluating two such strategies: (1) redefine $A_{exp}$ to include different expected availabilities for representative (ranges of) system parameters (e.g., numbers of components, hosts, location and collocation constraints, network reliabilities, and so on); and (2) select the algorithm that has been used most frequently in the past. We are comparing all three strategies for efficiency and overall achieved availability.

## 7. Effecting Redeployment

The output of each one of the algorithms is a desired deployment architecture (in the form of unique component-host identifier pairs), which now needs to be effected. The *DeployerComponent* controls this process as follows:

1. The *DeployerComponent* sends events to inform *AdminComponent*s of their new local configurations, and of the remote locations of software components required for performing changes to each local configuration.
2. Each *AdminComponent* determines the difference between its current and new configurations, and issues a series of events to remote *AdminComponent*s requesting the 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 relevant request events are sent to the *DeployerComponent*. The *DeployerComponent* then forwards those events to the appropriate destinations, and forwards the responses containing the migrant components to the requesting *AdminComponent*. Therefore, the *DeployerComponent* serves as a router for devices that are not directly connected.
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, 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.

As discussed in Section 4.2, our current implementation assumes a centralized organization, i.e., that the device containing the *DeployerComponent* will have direct connections with all the remaining devices. We are currently implementing and evaluating an existing decentralized solution to a similar problem [2].

To address the situations where the network reliability and bandwidth, and component sizes prevent atomic exchange of a migrant component between a pair of hosts (i.e., in a single serialized event), Prism-MW supports incremental component migration, in multiple events containing numbered byte stream segments of the component. After the last segment is sent, the sending *AdminComponent* issues a special event denoting that the process is complete. The receiving *AdminComponent* then reconstitutes the migrant component from the received segments, requesting that the sending *AdminComponent* resend any missing segments.

The time required to migrate a single component depends on the component's size, and reliability and bandwidth of the network link between the source and destination hosts. For example, if two hosts are connected by a low bandwidth dial-up connection of 56 Kbps, with reliability of 50%, a 10 KB component would get migrated in 2.9s. However, most infrared, wireless, and wired LAN links have a bandwidth that is at least an order of magnitude higher, which would reduce the migration time of the same component to under 1s. The time "cost" of the redeployment process $T_R$ corresponds to the size of a "dip" in the availability function in Figures 3 and 6. The upper bound of time, $T_R$, required to effect a system's deployment can be estimated using the following formula:

$$T_R = \sum_{i=1}^{n} T_R(i)$$

$$T_R(i) = \frac{mem_{comp}(i)}{bw(source(i), dest(i)) * rel(source(i), dest(i))} \quad if \quad bw(source(i), dest(i)) \neq 0$$

$$T_R(i) = \frac{mem_{comp}(i)}{bw(source(i), CH) * rel(source(i), CH)} + \frac{mem_{comp}(i)}{bw(dest(i), CH) * rel(dest(i), CH)} \quad otherwise$$

where *source(i)* is a source host of component *i*, *dest(i)* is the destination host of component *i*, *CH* is the host containing *DeployerComponent*, and *bw* is the function representing network bandwidth between a pair of hosts, defined as:

$$bw(h_i, h_j) = \begin{cases} \infty \quad if \quad h_i = h_j \\ 0 \quad if \quad h_i \ \text{is not connected to} \ h_j \\ \text{bandwidth of the link between } h_i \text{ and } h_j \quad if \quad h_i \neq h_j \end{cases}$$

For simplicity, the above formula does not include net-

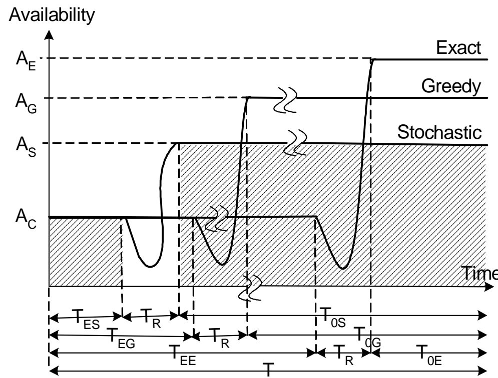


**Figure 6.** Selection criteria for algorithms.

work latency introduced by different network protocols (e.g., TCP-IP). Existing network latency estimation techniques (e.g., [4]) can be applied to more accurately estimate $T_R$. Finally, note that migrations on different hosts may happen in parallel, thus (significantly) reducing the estimated value of $T_R$.

## 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 [15]), often requiring significant involvement by the system's (human) operators. 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 enabling a distributed, mobile system to improve its availability via redeployment. Our support for disconnected operation has been successfully tested on several example applications [10,12]. 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 network configurations such as star, ring, and grid), (2) supporting decentralized redeployment, and (3) addressing the issue of trust in performing distributed redeployment. 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). In turn, this will likely result in added complexity of a system's decision making and self-reconfiguration code. We will study the applicability of existing solutions to this task.

## 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.


## 10. References


[1] M. C. Bastarrica, et.al. A Binary Integer Programming Model for Optimal Object Distribution. *2nd Int'l Conf. on Principles of Distributed Systems*, Amiens, France, December 1998.

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

[3] R. Grimes. *Professional DCOM Programming.* Wrox Press Inc., Chicago, IL, 1997.

[4] K. Gummadi, et. al. King: Estimating Latency between Arbitrary Internet End Hosts. *ACM SIGCOMM Computer Communication Review*, Vol. 32/3, July 2002.

[5] R. Haas et. al. Autonomic Service Deployment in Networks. *IBM Systems Journal*, Vol. 42, No. 1, 2003.

[6] IEEE Standard Computer Dictionary: *A Compilation of IEEE Standard Computer Glossaries.* New York, NY: 1990.

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

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

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

[10] N. Medvidovic, et al. Software Architectural Support for Handheld Computing. *IEEE Computer*, September 2003.

[11] M. Mikic-Rakic, et. al. Improving Availability in Large, Distributed, Component-Based Systems via Redeployment. Technical Report *USC-CSE-2003-515*, 2003.

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

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

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

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

[16] 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.