Improving Availability of Distributed Event-Based Systems
via Run-Time Monitoring and Analysis

Marija Mikic-Rakic, Sam Malek, Nels Beckman, and Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
{marija,malek,nbeckman,neno}@usc.edu

Abstract

A distributed software system's deployment architecture can have a significant impact on the system's availability, which depends on various system parameters, such as network bandwidth, frequencies of software component interactions, and so on. Existing system deployment tools lack support for monitoring, visualizing, and analyzing different factors that influence availability. They also lack support for modifying a running system's deployment to improve its availability. In this paper, we present an approach for runtime assessment of relevant system parameters, their visualization, and estimation and effecting of deployment architectures for large-scale, highly distributed systems.

1 Introduction

For any large, distributed system, many deployment architectures (i.e., distributions of the system's software components onto its hardware hosts) will be typically possible. Some of those deployment architectures will be more effective than others in terms of the desired system characteristics such as scalability, evolvability, mobility, and dependability. Availability is an aspect of dependability, defined as the degree to which the system is operational and accessible when required for use [4]. In the context of distributed environments, where a most common cause of (partial) system inaccessibility is network failure, we quantify availability as the ratio of the number of successfully completed inter-component interactions in the system to the total number of attempted interactions over a period of time. In other words, availability in distributed systems is greatly affected by the properties of the network, including its reliability and bandwidth.

Maximizing the availability of a given system may thus require the system to be redeployed such that the most critical, frequent, and voluminous interactions occur either locally or over reliable and capacious network links. However, finding the actual deployment architecture that maximizes a system's availability is an exponentially complex problem that may take *years* to resolve for any but very small systems [8]. Also, even a deployment architecture that increases the system's current availability by a desired amount cannot be easily found because of the *many* parameters that influence this task: number of hardware hosts, available memory and CPU power on each host, network topology, capacity and reliability of network links, number of software components, memory and processing require-

ments of each component, their configuration (i.e., software topology), frequency and volume of interaction among the components, and so forth. For these reasons, support for monitoring and visualizing relevant system parameters, estimating the deployment architecture based on these parameters in a manner that produces the desired (increase in) availability, and automatic effecting of the estimated deployment architecture is required.

In this paper we describe a three-stage approach to addressing the redeployment problem: (1) capturing and displaying the monitoring data from a running distributed system; (2) visualizing and exploring the system's deployment architecture; and (3) automatically updating the system's deployment architecture. To this end, we have combined capabilities of a visual exploration tool, DeSi [6] and an architectural middleware, Prism-MW [7]. DeSi supports specification, manipulation, visualization, and (re)estimation of deployment architectures for large-scale, highly distributed systems, while Prism-MW provides runtime facilities for monitoring a distributed system to assess the relevant system parameters, as well as support for effecting the desired redeployment architecture. Our support has been successfully evaluated on several examples.

The remainder of the paper is organized as follows. Section 2 defines the problem of increasing the availability of distributed systems, and outlines our approach. Section 3 presents an overview of DeSi and Prism-MW. Sections 4, 5, and 6 present the three stages of the redeployment process outlined above. The paper concludes with overviews of related and future work.

2 Problem and Approach

We describe a distributed system as:
- a set of *n* components with their properties: memory of each component and its frequencies of interaction with other components,
- a set of *k* hosts with their properties: available memory on each host, and its reliability of connectivity with other hosts, and
- a set of constraints that a valid deployment architecture must satisfy (e.g., the sum of memories of the components that are deployed onto a given host may not exceed the available memory on that host).

We define a system's availability *A* as the ratio of the number of successfully completed interactions in the system to the total number of attempted interactions, as fol-

lows:

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

where function f denotes the system's deployment architecture (i.e., $f(c_i) = h_j$ denotes that component c_i is deployed on host h_j). Function $freq$ captures the frequency of interaction among a pair of components, and function rel captures network reliability between a pair of hosts. The goal is to find a valid deployment f such that the system's availability is maximized.[1] In the most general case, the number of possible deployments is k^n. However, some of these deployments may be invalid (i.e., they may not satisfy the required constraints).

Our approach, illustrated in Figure 2, employs runtime redeployment to increase a system's availability by (1) monitoring the system, (2) visualizing, estimating, and analyzing its redeployment architecture, and (3) effecting the selected redeployment architecture. We leverage an architectural middleware, Prism-MW, to support runtime system monitoring. The monitoring information is then provided to our DeSi tool for system visualization and analysis. We have developed several algorithms that analyze and estimate improvements in deployment architectures [5,8]. Finally, after using DeSi to select the desired deployment architecture, Prism-MW facilities are leveraged to effect the selected deployment architecture.

3 Foundation

DeSi [6] is a visual deployment exploration environment that supports specification, manipulation, visualization, and (re)estimation of deployment architectures for large-scale, highly distributed systems. DeSi allows an engineer to rapidly explore the space of possible deployments for a given system (real or postulated), determine the deployments that will result in greatest improvements in availability (while, perhaps, requiring the smallest changes to the current deployment architecture), and assess a system's sensitivity to and visualize changes in specific parameters (e.g., the reliability of a network link) and deployment constraints (e.g., two components must be located on different hosts). DeSi allows one to easily integrate, evaluate, and compare different algorithms targeted at improving system availability [8] in terms of their feasibility, efficiency, and precision. As will be detailed in the remainder of this section, we have extended DeSi to allow its integration with any distributed middleware platform that supports system monitoring and runtime component deployment.

Prism-MW [7] is an extensible middleware platform, that enables efficient implementation, deployment, and execution of distributed software systems in terms of their architectural elements: components, connectors, configurations, and events [10]. For brevity, Figure 1 shows the sim-

plified class design view of Prism-MW. *Brick* is an abstract class that encapsulates common features of its subclasses (*Architecture, Component,* and *Connector*). The *Architecture* class records the configuration of its components and connectors, and provides facilities for their addition, removal, and reconnection, possibly at system runtime. A distributed application is implemented as a set of interacting *Architecture* objects, communicating via *DistributionConnector*s across process or machine boundaries. *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 probing of a *Brick*'s behavior, via different implementations of the *IMonitor* interface.

To support various aspects of architectural self-awareness, we have provided the *ExtensibleComponent* class, which contains a reference to *Architecture*. 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 runtime changes to the system's architecture.

In support of monitoring and redeployment, the *ExtensibleComponent* is augmented with the *IAdmin* interface. 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 interfacing with DeSi. 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 1, both *AdminComponent* and *DeployerComponent* contain a reference to *Architecture* and are thus able to effect runtime 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
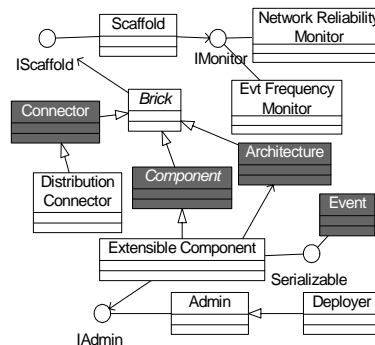

Figure 1. Simplified UML class design view of Prism-MW. The four dark gray classes are used by the application developer. Only the relevant middleware classes are shown.

---

1. A detailed description of the redeployment problem is given in [8].

between address spaces using the *Serializable* interface).

In order to perform runtime 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 (see Figure 2). One of the hosts contains the *DeployerComponent* (instead of the *AdminComponent*), which maintains a complete model of the system's current deployment architecture and interacts with DeSi.

To integrate DeSi with Prism-MW, we have wrapped DeSi as a Prism-MW component that is capable of receiving *Event*s with the monitoring data from Prism-MW's *DeployerComponent*, and issuing events to the *DeployerComponent* to enact a new deployment architecture. Once the monitoring data is received, DeSi updates its own system model. This results in the visualization of an actual system, which can now be analyzed and its deployment improved by employing different algorithms. Once the outcome of an algorithm is selected, DeSi issues a series of events to Prism-MW's *DeployerComponent* to update the system's deployment architecture.

The approach described in this paper assumes a centralized organization, i.e., that the device containing the *DeployerComponent* will have direct connections with all the remaining devices. As a part of our on-going work we
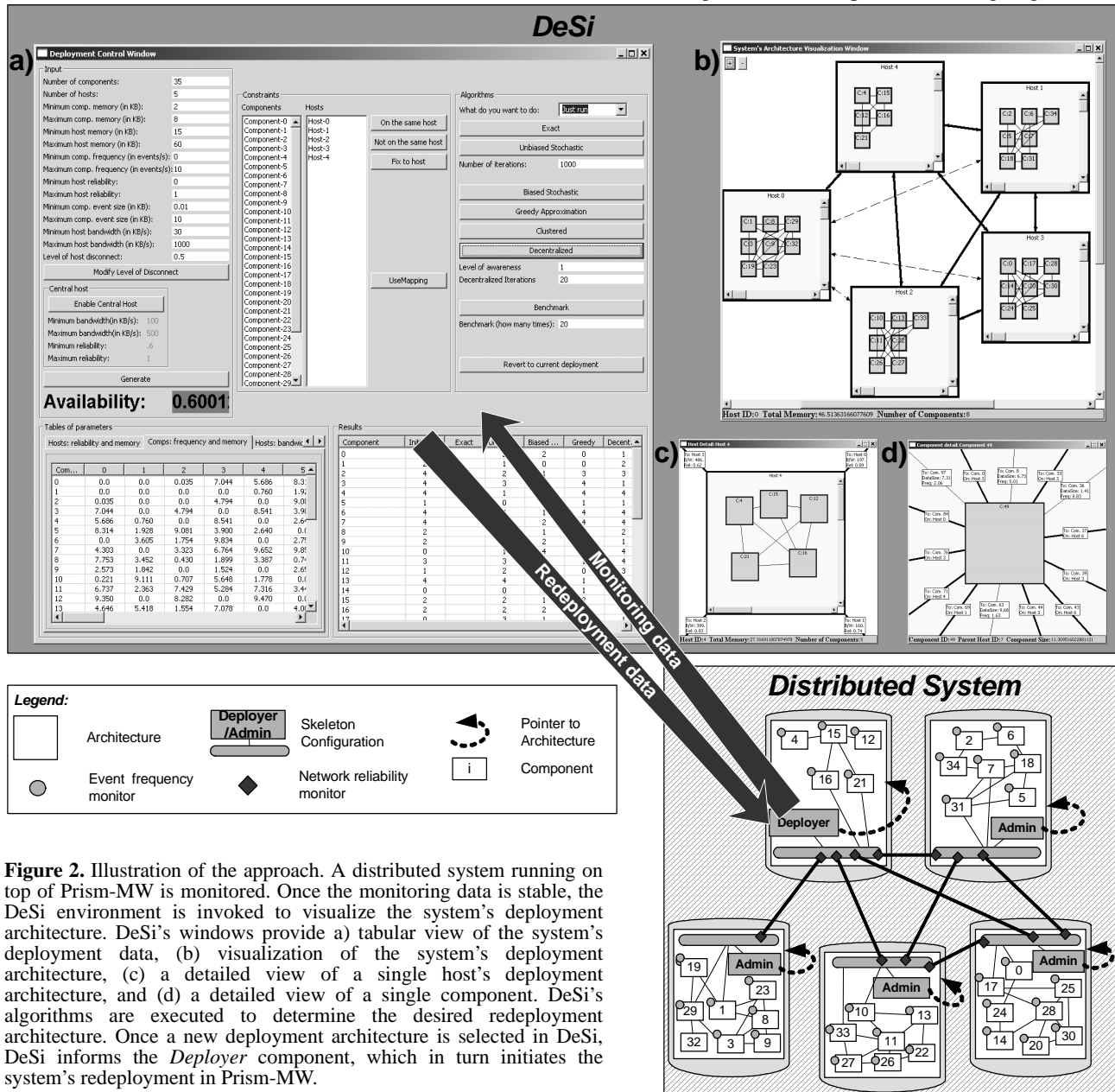

Figure 2. Illustration of the approach. A distributed system running on top of Prism-MW is monitored. Once the monitoring data is stable, the DeSi environment is invoked to visualize the system's deployment architecture. DeSi's windows provide a) tabular view of the system's deployment data, (b) visualization of the system's deployment architecture, (c) a detailed view of a single host's deployment architecture, and (d) a detailed view of a single component. DeSi's algorithms are executed to determine the desired redeployment architecture. Once a new deployment architecture is selected in DeSi, DeSi informs the *Deployer* component, which in turn initiates the system's redeployment in Prism-MW.

are extending Prism-MW to support decentralized system redeployment [5].

4 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 *NetworkReliabilityMonitor* records the reliability of connectivity between its associated *DistributionConnector* and other, remote *DistributionConnector*s using a common "pinging" technique.[2]

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 *Architecture*. The *AdminComponent* can also determine the memory size of the local architecture, which denotes the available memory on the corresponding host, via the reference to the *Architecture*. Through the same reference, the *AdminComponent* can record the memory size of each component within the architecture (e.g., by serializing the component into a byte array and determining the size of that array). The *AdminComponent* then sends the description of its local deployment architecture (i.e., local configuration) and the monitoring data (i.e., memory, event frequency, and network reliability) in the form of serialized *Event*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* 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 ε), the *AdminComponent* assumes that the monitoring data is stable, and informs the *DeployerComponent*.

An issue we have considered deals with cases when most, but not all system parameters are stable. As described above, the monitoring data is not considered stable until all the parameters satisfy their ε constraint. There are at least two ways of addressing this situation. The first is to increase the ε for the "troublesome" parameters. Alternatively, a single, global ε_g may be used to assume stability, as soon as the average difference of the monitoring data for all the parameters in a single, local *Architecture* becomes smaller than ε_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.

5 System Visualization and Analysis

The *DeployerComponent* accesses its local monitoring data; it also receives all the monitoring data and local configuration data from the remote *AdminComponent*s. Once the monitoring data is gathered from all the hosts, the *DeployerComponent* invokes the DeSi environment to visualize the system's deployment architecture and its relevant parameters.

Figure 2a shows DeSi's main window, which displays a distributed system's parameters (in the *Tables of parameters* panel). In the *Constraints* panel, the user can specify different constraints on component locations (e.g., fixing a component to a selected host). Using the set of buttons in the *Algorithms* panel, different algorithms [5,8] can be invoked and the results displayed in the *Results* panel. Finally, the user can modify the desired system parameters (by editing the appropriate tables in *Tables of parameters* panel) to assess the sensitivity of a deployment architecture to parameter changes.

Figure 2b shows the graphical display of the deployment architecture of a system with 35 components and 5 hosts in DeSi (i.e., the visualization of the architecture as extracted by Prism-MW's *AdminComponent*s and *DeployerComponent*). Network connections between hosts are depicted as solid lines, while dotted lines between pairs of hosts denote that some of the components on the two respective hosts need to communicate, but that there is no network link between them. In order to support effective visualizations of large distributed deployment architectures, DeSi supports zooming in and out, and provides the ability to "drag" hosts and components on-screen, in which case all relevant links will follow them. Detailed information about a host or component can be displayed by double-clicking on the corresponding graphical object. The detailed information for a host, shown in Figure 2c, displays the host's properties in the status bar, the components deployed on the host, the host's connections to other hosts, and the reliabilities of those connections. Similarly, the detailed information for a component, shown in Figure 2d, displays the component's properties and its connections to other components.

DeSi's combination of different system visualizations, redeployment algorithms, and exploration capabilities results in a rich and intuitive environment for analyzing and improving a system's deployment. After performing the analysis, the user selects an algorithm's result, i.e., a desired deployment architecture (recall function *f* in Section 2), which now needs to be effected.

---

2. Note that Prism-MW's extensibility [7] can be easily leveraged to support monitoring of other system properties (e.g., network bandwidth, volume of exchanged data, network latency).

6 Effecting Redeployment

DeSi environment informs the *DeployerComponent* of the desired deployment architecture (via a Prism-MW *Event* containing unique component-host identifier pairs), which now needs to be effected. The *DeployerComponent* controls the redeployment process as follows:

1. The *DeployerComponent* sends events to inform *Admin-Component*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.

7 Related Work

We have performed an extensive survey of existing approaches aimed at improving system's availability in face of network failures, and provided a framework for their classification and comparison [9]. One of the techniques for improving system availability is (re)deployment, which is a process of installing, updating, and/or relocating a distributed software system.

Carzaniga et. al. [1] provide an extensive comparison of existing *software deployment* approaches. They identify several issues lacking in the existing deployment tools, including integrated support for the entire deployment life-cycle. An exception is Software Dock [2], which is a system of loosely coupled, cooperating, distributed components. It provides software deployment agents that travel among hosts to perform software deployment tasks. Unlike our approach, however, Software Dock does not focus on extracting system parameters, visualizing, or evaluating a system's deployment architecture.

Finally, Haas et. al. [3] provide a scalable framework for autonomic service deployment in networks. This approach does not address the inherent exponential complexity in the selection of the most appropriate deployment, or that prop-erties of services and hosts may change during the system's execution.

8 Conclusion

A distributed software system's deployment architecture can have a significant impact on the system's availability, and will depend on various system parameters (e.g., reliability of connectivity among hosts, security of links between hosts, and so on). Existing deployment approaches focus on providing support for installing and updating the software system but lack support for extracting, visualizing, and analyzing different parameters that influence the quality of deployment.

This paper has presented an integration of Prism-MW, a lightweight architectural middleware that supports system monitoring and runtime reconfiguration, and DeSi, an environment that supports manipulation, visualization, and (re)estimation of deployment architectures for large-scale, highly distributed systems. In concert, Prism-MW and DeSi provide a rich capability for improving the availability of distributed systems. Although our experience to date has been very positive, it has suggested a number of possible avenues for further work. We plan to address issues such as supporting decentralized redeployment, and addressing the issue of trust in performing distributed redeployment. Finally, we plan to extend DeSi's analysis capabilities to automatically determine the sensitivity of a deployment architecture to variations of system parameters. The results of this analysis would be used to inform Prism-MW's monitoring of what constitutes a significant change in system parameters (i.e., the values of ε for each parameter).

9 References

[1] A. Carzaniga et. al. A Characterization Framework for Software Deployment Technologies. *Technical Report*, Dept. of Computer Science, University of Colorado, 1998.
[2] R. S. Hall, D. Heimbigner, and A. L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. *ICSE'99*, Los Angeles, CA, May 1999.
[3] R. Haas et. al. Autonomic Service Deployment in Networks. *IBM Systems Journal*, Vol. 42, No. 1, 2003.
[4] IEEE Standard Computer Dictionary: *A Compilation of IEEE Standard Computer Glossaries.* New York, NY: 1990.
[5] S. Malek et. al. A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems. Technical Report *USC-CSE-2004-506*, 2004.
[6] M. Mikic-Rakic et. al. A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings. To Appear in *2nd Int'l Working Conf. on Component Deployment (CD 2004)*, Edinburgh, Scotland, May 2004.
[7] M. Mikic-Rakic and N. Medvidovic. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. *Middleware 2003*, Rio De Janeiro, June 2003.
[8] M. Mikic-Rakic and N. Medvidovic. Support for Disconnected Operation via Architectural Self-Reconfiguration. *Int'l Conf. on Autonomic Computing (ICAC'04)*, New York, May 2004.
[9] M. Mikic-Rakic and N. Medvidovic. Toward a Framework for Classifying Disconnected Operation Techniques. *ICSE WADS'03*, Portland, Oregon, May 2003.
[10] D.E. Perry, and A.L. Wolf. Foundations for the Study of Software Architectures. *Software Engineering Notes*, Oct. 1992.