Wolfgang Emmerich
Alexander L. Wolf (Eds.)

LNCS 3083

Component Deployment

Second International Working Conference, CD 2004
Edinburgh, UK, May 2004
Proceedings

Springer

TP311.5-53
C737
2004

Wolfgang Emmerich   Alexander L. Wolf (Eds.)

Component Deployment

Second International Working Conference, CD 2004
Edinburgh, UK, May 20-21, 2004
Proceedings

Springer

Volume Editors

Wolfgang Emmerich
University College London
Dept. of Computer Science
Gower Street, London WC1E 6BT, UK
E-mail: w.emmerich@cs.ucl.ac.uk

Alexander L. Wolf
University of Colorado
Department of Computer Science
Boulder, Colorado, 80309-430 USA
E-mail: alw@cs.colorado.edu

Library of Congress Control Number: 2004105536

CR Subject Classification (1998): D.2, F.3, D.1, D.3, D.4

ISSN 0302-9743
ISBN 3-540-22059-3 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 11009528 06/3142 5 4 3 2 1 0

Preface

This volume of the Lecture Notes in Computer Science series contains the proceedings of the second Working Conference on Component Deployment, which took place May 20–21, 2004, at the e-Science Institute in Edinburgh, Scotland, as a collocated event of the International Conference on Software Engineering.

Component deployment addresses what needs to be done *after* a component has been developed. Component deployment includes activities such as component customization, configuration, integration, activation, de-activation and decommissioning. The emerging research community that investigates component deployment concerns itself with the principles, methods and tools for deployment activities. The community held its first working conference in Berlin, Germany, in June 2002. The proceedings were published by Springer-Verlag as volume 2370 of the Lecture Notes in Computer Science series.

The program of this year's conference consisted of an invited talk and 16 technical paper presentations. The invited talk was given by Patrick Goldsack of Hewlett Packard Research Laboratories Bristol, UK. He presented the SmartFrog component deployment framework that HP released as Open Source. The technical papers were carefully selected from a total of 34 submitted papers. Each paper was thoroughly peer reviewed by at least three members of the program committee and consensus on acceptance was achieved by means of an electronic PC meeting.

The conference and these proceedings would not have been possible without the help of a large number of people. Anthony Finkelstein, in his role as General Chair of ICSE, simplified our task considerably by arranging our use of the CyberChair electronic submission and reviewing service, as well as handling publicity and registration. We are indebted to ACM SIGSOFT and the UK e-Science Programme for generously providing support for the conference, and to Malcolm Atkinson and Dave Berry at the e-Science Institute for hosting CD 2004. Particular thanks go to Gill Mandy for handling the local arrangements. Richard van der Stadt of Borbala was always available and responded incredibly quickly whenever we needed him and, as a result, he eased the paper submission and review process considerably. Finally, we thank the members of the program committee for their hard work and careful reviews.

March 2004 Wolfgang Emmerich and Alexander L. Wolf

Program Committee

Sponsoring Institutions

ACM Special Interest Group on Software Engineering (SIGSOFT)
UK e-Science Programme

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Table of Contents

A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings

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 properties. These properties will depend on various system parameters, such as network bandwidth, frequencies of software component interactions, and so on. Existing tools for representing system deployment lack support for specifying, visualizing, and analyzing different factors that influence the quality of a deployment, e.g., the deployment's impact on the system's availability. In this paper, we present an environment that supports flexible and tailorable specification, manipulation, visualization, and (re)estimation of deployment architectures for large-scale, highly distributed systems. The environment has been successfully used to explore large numbers of postulated deployment architectures. It has also been integrated with a middleware platform to support the exploration of deployment architectures of actual distributed systems.

Keywords. Software deployment, availability, disconnection, visualization, environment, middleware

1 Introduction

For any large, distributed system, multiple deployment architectures (i.e., distributions of the system's software components onto its hardware hosts, see Fig. 1.) 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 [5]. In the context of distributed environments, where a most common cause of (partial) system inaccessibility is network failure [17], 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 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 [11]. 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 requirements of each component, their configuration (i.e., software topology), frequency and volume of interaction among the components, and so forth. A naive solution to this problem would be to keep redeploying the actual system that exhibits poor availability until an adequate deployment architecture is found. However, this would be prohibitively expensive. A much more preferable solution is to develop a means of modeling the relevant system parameters, estimating the deployment architecture based on these parameters in a manner that produces the desired (increase in) availability, and assessing the estimated architecture in a controlled setting, *prior* to changing the actual deployed system.

In this paper, we discuss a tailorable environment developed precisely for that purpose. The environment, called DeSi, 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 particular network link) and deployment constraints (e.g., two components must be located on different hosts). We have provided a facility that automatically generates large numbers of deployment scenarios and have evaluated different aspects of DeSi using this facility. DeSi also allows one to easily integrate, evaluate, and compare different algorithms targeted at improving system availability [11] in terms of their feasibility, efficiency, and precision. We illustrate this support by showing the integration of six such algorithms.

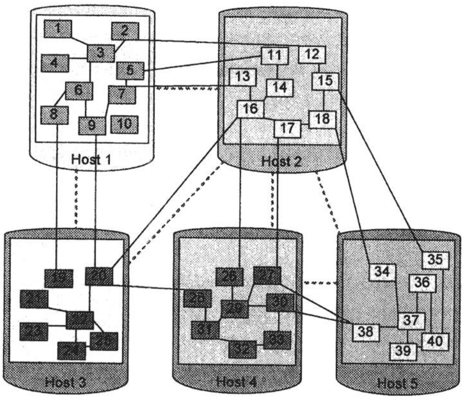

Fig. 1. Example deployment architecture: A software system comprising 40 components is deployed onto five hosts. The dotted lines represent host interconnectivity; filled lines represent software component interaction paths

DeSi also provides a simple API that allows its integration with any distributed system platform (i.e., middleware) that supports component deployment at runtime. We demonstrate this support by integrating DeSi with the Prism-MW middleware [10]. Finally, while availability has been our focus to date, DeSi's architecture is flexible enough to allow exploration of other system characteristics (e.g., security, fault-tolerance, and so on).

The remainder of the paper is organized as follows. Section 2 defines the problem of increasing the availability of distributed systems, and overviews six different algorithms we have developed for this purpose. Section 3 highlights the related work. Section 4 discusses the architecture, implementation, and usage of the DeSi environment. Evaluation of DeSi is presented in Section 5. The paper concludes with a discussion of future work.

2 Background

2.1 Problem Description

The distribution of software components onto hardware nodes (i.e., a system's software *deployment architecture*, illustrated in Fig. 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; components distributed across different hosts might not. However, the reliability (i.e., rate of failure) of connectivity among the hardware nodes on which the system is deployed may not be 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 current 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.

In addition to the characteristics of hardware connectivity and software interaction, there are other constraints on a system's redeployment, including the available memory on each network host, the required memory for each software component, the size of data exchanged between software components, the bandwidth of each network link, and possible restrictions on component locations (e.g., a component may be fixed to a selected host, or two components may not be allowed to reside on the same host). Fig.2 shows a formal model that captures the system properties and constraints, and a formal definition of the problem we are addressing. 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, and the average size of data exchanged between them is captured via the *evt_size* 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, and the network bandwidth via the *bw* function. Using the *loc* function, deployment of any component can be restricted to a subset of hosts, thus denoting a set of allowed hosts for that component. Using the *colloc* function, constraints on collocation of components can be specified.

The definition of the problem contains the criterion function *A*, which formally describes a system's availability as the ratio of the number of successfully completed interactions in the system to the total number of attempted interactions. Function *f* represents the exponential number of the system's candidate deployments. To be considered valid, each candidate deployment must satisfy the four stated conditions. The first condition states that the sum of memories of the components deployed onto a given host may not exceed the host's available memory. The second condition states that the total volume of data exchanged across any link between two hosts may not exceed the link's *effective bandwidth*, which is the product of the link's actual bandwidth and its reliability. The third condition states that a component may only be deployed onto a host that belongs to a set of allowed hosts for that component,

Model

Given:

(1) a set C of n components ($n=|C|$) and three functions $freq: C\times C \rightarrow \Re$, $evt\_size: C\times C \rightarrow \Re$, and $mem_{comp}: C \rightarrow \Re$

$$freq(c_i,c_j)=\begin{pmatrix} 0 \quad if \quad c_i=c_j \\ frequency\ of\ communication\ between\ c_i\ and\ c_j \quad if \quad c_i\neq c_j \end{pmatrix} \quad evt\_size(c_i,c_j)=\begin{pmatrix} 0 \quad if \quad c_i=c_j \\ avg\ size\ of\ data\ c_i\ and\ c_j\ exchange \quad if \quad c_i\neq c_j \end{pmatrix}$$

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

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

$$rel(h_i,h_j)=\begin{pmatrix} 1 \quad if \quad h_i=h_j \\ 0 \quad if \quad h_i\ is\ not\ connected\ to\ h_j \\ reliability\ of\ the\ link\ between\ h_i\ and\ h_j \quad if \quad h_i\neq h_j \end{pmatrix} \quad bw(h_i,h_j)=\begin{pmatrix} \infty \quad if \quad h_i=h_j \\ 0 \quad if \quad h_i\ is\ not\ connected\ to\ h_j \\ bandwidth\ of\ the\ link\ between\ h_i\ and\ h_j \quad if \quad h_i\neq h_j \end{pmatrix}$$

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

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

$$loc(c_i,h_j)=\begin{pmatrix} 1 \quad if \quad c_i\ can\ be\ deployed\ onto\ h_j \\ 0 \quad if \quad c_i\ cannot\ be\ deployed\ onto\ h_j \end{pmatrix} \quad colloc(c_i,c_j)=\begin{pmatrix} -1 \quad if \quad c_i\ cannot\ be\ on\ the\ same\ host\ as\ c_j \\ 1 \quad if \quad c_i\ has\ to\ be\ on\ the\ same\ host\ as\ c_j \\ 0 \quad if \quad there\ are\ no\ restrictions\ on\ collocation\ of\ c_i\ and\ c_j \end{pmatrix}$$

Problem

Problem:

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

A defined as $A=\dfrac{\sum_{i=1}^{n}\sum_{j=1}^{n}(freq(c_i,c_j)*rel(f(c_i),f(c_j)))}{\sum_{i=1}^{n}\sum_{j=1}^{n}freq(c_i,c_j)}$ is maximized, and the following four conditions are satisfied:

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

(2) $(\forall i\in[1,k] \quad \forall j\in[i+1,k]) \begin{pmatrix} (\forall l\in[1,n] \quad \forall m\in[l+1,n]) \\ \begin{pmatrix} where\ f(c_l)=h_i \wedge f(c_m)=h_j \\ \sum_{l,m} data\_vol(c_l,c_m) \leq effective\_bw(h_i,h_j) \end{pmatrix} \end{pmatrix}$ where *data_vol* and *effective_bw* are defined as follows:

$$data\_vol(c_x,c_y)=freq(c_x,c_y)*evt\_size(c_x,c_y) \quad effective\_bw(h_x,h_y)=rel(h_x,h_y)*bw(h_x,h_y)$$

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

(4) $\forall i\in[1,n] \quad \forall j\in[i+1,n] \qquad (colloc(c_i,c_j)=1) \Rightarrow (f(c_i)=f(c_j)) \quad (colloc(c_i,c_j)=-1) \Rightarrow (f(c_i)\neq f(c_j))$

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 four conditions.

Fig. 2. Formal statement of the problem

specified via the *loc* function. Finally, the fourth condition states that two components must be deployed onto the same host (or on different hosts) if required by the *colloc* function.

2.2 Algorithms

In this section we briefly describe six algorithms we have developed for increasing a system's availability by calculating a new deployment architecture. A detailed performance comparison of several of these algorithms is given in [11].

Exact Algorithm: This algorithm tries every possible deployment, and selects the one that has maximum availability and satisfies the constraints posed by the memory, bandwidth, and restrictions on software component locations. The exact algorithm guarantees at least one optimal deployment (assuming that at least one deployment is possible). The complexity of this algorithm in the general case (i.e., with no restrictions on component locations) is $O(k^n)$, where k is the number of hardware hosts, and n the number of software components. By fixing a subset of m components to selected hosts, the complexity reduces to $O(k^{n-m})$.

Unbiased Stochastic Algorithm: This algorithm generates different deployments by randomly assigning each component to a single host from the set of available hosts for that component. If the randomly generated deployment satisfies all the constraints, the availability of the produced deployment architecture is calculated. This process repeats a given number of times and the deployment with the best availability is selected. As indicated in Fig. 2, the complexity of calculating the availability for each valid deployment is $O(n^2)$, resulting in the same complexity of the overall algorithm.

Biased Stochastic Algorithm: This algorithm randomly orders all the hosts and all the components. Then, going in order, it assigns as many components to a given host as can fit on that host, ensuring that all of the constraints are satisfied. Once the host is full, the algorithm proceeds with the same process for the next host in the ordered list of hosts, and the remaining unassigned components in the ordered list of components, until all components have been deployed. This process is repeated a desired number of times, and the best obtained deployment is selected. Since it needs to calculate the availability for every deployment, the complexity of this algorithm is $O(n^2)$.

Greedy Algorithm: This algorithm incrementally assigns software components to the hardware hosts. At each step of the algorithm, the goal is to select the assignment that will maximally contribute to the availability function, by selecting the "best" host and "best" software component. Selecting the best hardware host is performed by choosing a host with the highest sum of network reliabilities with other hosts in the system, and the highest memory capacity. Similarly, selecting the best software component is performed by choosing the component with the highest frequency of interaction with other components in the system, and the lowest required memory. Once found, the best component is assigned to the best host, making certain that the four constraints are satisfied. The algorithm proceeds with searching for the next best component among the remaining components, until the best host is full. Next, the algorithm selects the best host among the remaining hosts. This process repeats until every component is assigned to a host. The complexity of this algorithm is $O(n^3)$ [**11**].

Clustering Algorithm: This algorithm groups software components and physical hosts into a set of component and host *clusters*, where all members of a cluster are treated as a single entity. For example, when a component in a given cluster needs to be redeployed to a new host, all of the cluster's member components are redeployed. The algorithm clusters components with high frequencies of interaction, and hosts with high connection reliability. Clustering can significantly reduce the size of the redeployment problem; it also has the potential to increase the availability of a system. For example, connectivity-based clustering in peer-to-peer networks improves the quality of service by reducing the cost of messaging [**15**].

Decentralized Algorithm: The above algorithms assume the existence of a central host with reliable connections to every other host in the system. This assumption does not hold in a wide range of distributed systems (e.g., ad-hoc mobile networks), requiring a decentralized solution. Our decentralized redeployment algorithm [**8**] leverages a variation of the auction algorithm, in which each hosts acts as an agent and may conduct or participate in auctions. Each host's agent initiates an auction for the redeployment of its local components, assuming none of its neighboring (i.e., connected) hosts is already conducting an auction. The auction initiation is done by sending to all the neighboring hosts a message that carries information about a component (e.g., name, size, and so on). The agents receiving this message have a limited time to enter a bid on the component before the auction closes. The bidding agent on a given host calculates an initial bid for the auctioned component, by considering the frequency and volume of interaction between components on its host and the auctioned component. In each bid message, the bidding agent also sends additional local information, including its host's network reliability and bandwidth with neighboring hosts. Once the auctioneer has received all the bids, it calculates the final bid based on the received information. The host with the highest bid is selected as the winner. If the winner has enough free memory and sufficient bandwidth to host the auctioned component, then the component is redeployed to it and the auction is closed. If this is not the case, then the winner and the auctioneer attempt to find a component on the winner host to be traded (swapped) with the auctioned component. The complexity of this algorithm is $O(k*n^3)$.

3 Related Work

This section briefly outlines several research areas and approaches relevant to our work on DeSi: software architectures, disconnected operation, software deployment, software visualization, and visual software environments.

Software architectures provide high-level abstractions for representing structure, behavior, and key properties of a software system [14]. They are described in terms of *components*, which describe the computations and state of a system; *connectors*, which describe the rules and mechanisms of interaction among the components; and *configurations*, which define topologies of components and connectors. DeSi leverages an architectural model of a distributed system, including its deployment information. In our approach, a component represents the smallest unit of deployment.

Disconnected operation refers to the continued functioning of a distributed system in the (temporary) absence of network connectivity. We have performed an extensive survey of existing disconnected operation approaches, and provided a framework for their classification and comparison [12]. One of the techniques for supporting disconnected operation is (re)deployment, which is a process of installing, updating, 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 lifecycle. An exception is Software Dock [4], which has been proposed as a systematic framework that provides that support. Software Dock is a system of loosely coupled, cooperating, distributed components. It provides software deployment agents that travel among hosts to perform software deployment tasks. Unlike DeSi, however, Software Dock does not focus on visualizing, automatically selecting, or evaluating a system's deployment architecture.

UML [13] is the primary notation for the *visual modeling* of today's software systems. UML's deployment diagram provides a standard notation for representing a system's software deployment architecture. Several recent approaches extend this notation via stereotypes [3,7]. However, using UML to visualize deployment architectures has several drawbacks: UML's deployment diagrams are static; they do not depict connections among hardware hosts; and they do not provide support for representing and visualizing the parameters that affect the key system properties (e.g., availability). For these reasons, we have opted not to use a UML-based notation in DeSi.

There are several examples of *visual software development environments* that have originated from industrial and academic research. For example, AcmeStudio [16] is an environment for modeling, visualizing, and analyzing software architectures. Environments such as Visual Studio [9] provide a toolset for rapid application development, testing, and packaging. In our context, the role of the DeSi environment is to support tailorable, scalable, and platform-independent modeling, visualization, evaluation, and implementation of highly distributed systems. For these reasons we opted for using Eclipse [2] in the construction of DeSi. Eclipse is a platform-independent IDE for Java with support for plug-ins. Eclipse provides an efficient graphical library (Draw2D) and accompanying graphical editing framework (GEF), which we leveraged in creating visual representations of deployment architectures in DeSi.