Infrastructure for Automatic Dynamic Deployment of J2EE Applications in Distributed Environments

CIMS Technical Report: TR2005-867

Anatoly Akkerman, Alexander Totok, and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, NY, USA
{akkerman,totok,vijayk}@cs.nyu.edu

Abstract

Recent studies showed potential for using component frameworks for building flexible adaptible applications for deployment in distributed environments. However this approach is hindered by the complexity of deployment of component-based applications, which usually involves a great deal of configuration of both the application components and system services they depend on. In this paper we propose an infrastructure for automatic dynamic deployment of J2EE applications, that specifically addresses the problems of (1) inter-component connectivity specification and its effects on component configuration and deployment; and (2) application component dependencies on application server services, their configuration and deployment. The proposed infrastructure provides simple yet expressive abstractions for potential application adaptation through dynamic deployment and undeployment of components. We implement the infrastructure as a part of the JBoss J2EE application server and test it on several sample J2EE applications.

1 Introduction

In recent years, we have seen a significant growth in component-based enterprise application development. These applications are typically deployed on company Intranets or on the Internet and are characterized by high transaction volume, large numbers of users and wide area access. Traditionally they are deployed in a central location, using server clustering with load balancing (horizontal partitioning) to sustain user load. However, horizontal partitioning has been shown very efficient only in reducing application-related overheads of user-perceived response times, without having much effect on network-induced latencies. Vertical partitioning (e.g., running web tier and business tier in separate VMs) has been used for fault isolation and load balancing but it is sometimes impractical due to significant run-time overheads (even if one would keep the tiers on a fast local-area network) related to heavy use of remote invocations. Recent work [14] in the context of J2EE component-based applications has shown viability of vertical partitioning in wide-area networks without incurring the aforementioned overheads. The key conclusions from that study can be summarized as follows:

- Using properly designed applications, vertical distribution across wide-area networks improves user-perceived latencies.
- Wide-area vertical layering requires replication of application components and maintaining consistency between replicas.
- Additional replicas may be deployed dynamically to handle new requests.
- Different replicas may, in fact, be different implementations of the same component based on usage (read-only, read-write).
- New request paths may reuse components from previously deployed paths.

Applying intelligent monitoring [6] and AI planning [2, 12] techniques in conjunction with the conclusions of that study, we see a potential for dynamic adaptation in industry-standard J2EE component-based applications in wide area networks through deployment of additional application components dynamically based on active monitoring.

However, in order to achieve such dynamic adaptation, we need an infrastructure for automating J2EE application deployment in such an environment. This need is quite evident to anyone who has ever tried deploying a J2EE application even on a single application server, which is a task that involves a great deal of configuration of both the system services and application components. For example one has to set up JDBC data sources, messaging destinations and other resource adapters before application components can be configured and deployed. In a wide area deployment that spans multiple server nodes, this proves even more complex, since more system services that facilitate inter-node communications need to be configured and started and a variety of configuration data, like IP addresses, port numbers, JNDI names and others have to be consistently maintained in various configuration files on multiple nodes.

This distributed deployment infrastructure must be able to:

- address inter-component connectivity specification and define its effects on component configuration and deployment,
- address application component dependencies on application server services, their configuration and deployment,
- provide simple but expressive abstractions to control adaptation through dynamic deployment and undeployment of components,
- enable reuse of services and components to maintain efficient use of network nodes' resources,
- provide these facilities without incurring significant additional design effort on behalf of application programmers.

In this paper we propose the infrastructure for automatic dynamic deployment of J2EE applications, that addresses all of the aforementioned issues. The infrastructure defines architecture description languages (ADL) for component and link description and assembly. The *Component Description Language* is used to describe application components and links. It provides clear separation of application components from system components. A flexible type system is used to define compatibility of component ports and links. A declaration and expression language for *configurable component properties* allows for specification of inter-component dependencies and propagation of properties between components. The *Component (Replica) Assembly Language* allows for assembly of replicas of previously defined components into application paths by connecting appropriate ports via link replicas and specifying the mapping of these component replicas onto target application server nodes. The *Component Configuration Process* evaluates an application path's correctness, identifies the dependencies of application components on system components, and configures component replicas for deployment. An attempt is made to match and reuse any previously deployed replicas in the new path based on their configurations.

We implement the infrastructure as a part of the JBoss open source Java application server [11] and test it on several sample J2EE applications – Java PetStore [23], RUBiS [20] and TPC-W-NYU [32]. The infrastructure implementation utilizes the JBoss's extendable *micro-kernel* architecture, based on the JMX [27] specification. Componentized architecture of JBoss allows incremental service deployments depending on the needs of deployed applications. We believe that dynamic reconfiguration of application servers through dynamic deployment and undeployment of system services is essential to building a resource-efficient framework for dynamic distributed deployment of J2EE applications.

The rest of the paper is organized as follows. Section 2 provides necessary background for understanding the specifics of the J2EE component technology which are relevant to this study. Section 3 gives a general description of the infrastructure architecture, while section 4 goes deeper in describing particularily important and interesting internal mechanisms of the infrastructure. Section 5 describes the implementation of the framework, and related work is discussed in section 6.

2 J2EE Background

2.1 Introduction

Component frameworks. A component framework is a middleware system that supports applications consisting of components conforming to certain standards. Application components are "plugged" into the component framework, which establishes their environmental conditions and regulates the interactions between them. This is usually done through *containers*, component holders, which also provide commonly required support for naming, security, transactions, and persistence.

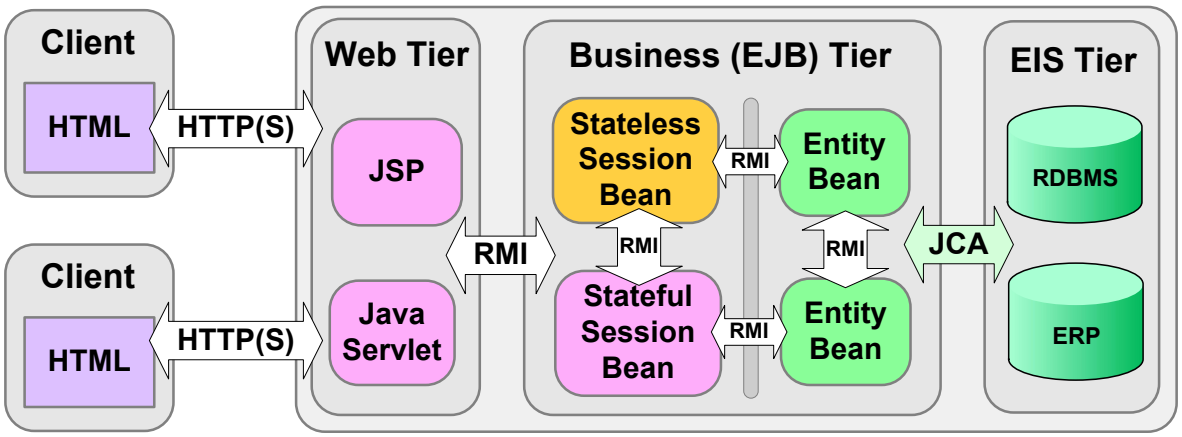

Figure 1: J2EE 3-Tier architecture.

Component frameworks provide an integrated environment for component execution, as a result significantly reduce the effort it takes to design, implement, deploy, and maintain applications. Current day industry component framework standards are represented by Object Management Group's CORBA Component Model [18], Sun Microsystems' Java 2 Platform Enterprise Edition (J2EE) [25] and Microsoft's .NET [17], with J2EE being currently the most popular and widely used component framework in the enterprise arena.

J2EE. Java 2 Platform Enterprise Edition (J2EE) [25] is a comprehensive standard for developing multi-tier enterprise Java applications. The J2EE specification among other things defines the following:

- component programming model,
- component contracts with the hosting server,
- services that the platform provides to these components,
- various human roles,
- compatibility test suites and compliance testing procedures.

Among the list of services that a compliant application server must provide are messaging, transactions, naming and others that can be used by the application components.

Application developed using J2EE adhere to the classical 3-Tier architechture – *Presentation Tier*, *Business Tier*, and *Enterprise Information System (EIS) Tier* (see Fig. 1). J2EE components belonging to each tier are developed adhering to the specific J2EE standards.

1. **Presentation or Web tier.**

   This tier is actually subdivided into client and server sides. The client side hosts a web browser, applets and Java applications that communicate with the server side of presentation tier or the business tier. The server side hosts Java Servlet components [30], Java Server Pages (JSPs) [29] and static web content. These components are responsible for presenting business data to the end users. The data itself is typically acquired from the business tier and sometimes directly from the Enterprise Information System tier. The server side of the presentation tier is typically accessed through HTTP(S) protocol.

2. **Business or EJB tier.**

   This tier consists of Enterprise Java Beans (EJBs) [24] that model the business logic of the enterprise application. These components provide persistence mechanisms and transactional support. The components in the EJB tier are invoked

through remote invocations (RMI), in-JVM invocations or asynchronous message delivery, depending on the type of EJB component.

The EJB specification defines several types of components. They differ in invocation style (synchronous vs. asynchronous, local vs. remote) and statefulness: completely stateless (e.g., Message-Driven Bean), stateful non-persistent (e.g., Stateful Session Bean), stateful persistent (e.g., Entity Bean). Synchronously invocable EJB components expose themselves through a special factory proxy object (an *EJB Home* object, which is specific to a given EJB), which is typically bound in JNDI by the deployer of the EJB. The EJB Home object allows creation or location of an *EJB Object*, which is a proxy to a particular instance of a this EJB [1].

3. **Enterprise Information System (EIS) or Data tier.**

   This tier refers to the enterprise information systems, like relational databases, ERP systems, messaging systems and the like. Business and presentation tier component communicate with this tier with the help of resource adapters as defined by the Java Connector Architecture [26].

The J2EE programming model has been conceived as a distributed programming model where application components would run in J2EE servers and communicate with each other. After the initial introduction and first server implementations, the technology, most notably, the EJB technology, has seen some a significant shift away from purely distributed computing model towards local interactions [2]. There were very legitimate performance-related reasons behind this shift, however the distributed features are still available.

The J2EE specification has seen several revisions, the latest stable being version 1.3, while version 1.4 is going through last review phases [3]. We shall focus our attention on the former, while actually learning from the latter.

Compliant commercial J2EE implementations are widely available from BEA Systems [4], IBM [9], Oracle [21] and other vendors. Several open source implementations, including JBoss [11] and JOnAS [19] claim compatibility as well. A recent addition to the list is a new Apache project Geronimo [1].

2.2 J2EE Component Programming Model

Before we describe basic J2EE components, let's first address the issue of defining what a component is.

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [31].

According to this definition the following entities which make up a typical J2EE application would be considered application components (some exceptions given below):

- EJBs (session, entity, message-driven),
- Web components (servlets, JSPs),
- messaging destinations,
- data sources,

EJB and Web components are deployed into their corresponding containers provided by the application server vendor. They have well-defined contracts with their containers that govern lifecycle, threading, persistence and other concerns. Both Web and EJB components use JNDI lookups to locate resources or other EJB components they want to communicate with. The JNDI context in which these lookups are performed is maintained separately for each component by its container. Bindings in this context are typically configured by the component's deployment descriptors.

[1] A deployment of an EJB component may contain multiple instances of an EJB. For example, during an online store application lifetime, a deployment of a stateful session bean that holds contents of an online shopping cart may consist of several instances of the shopping cart bean, one for each shopper currently shopping on the site. Similarly, an entity bean that holds persistent user account information has a unique instance corresponding to each account number on record. Please, refer to relevant sections in the EJB [24] and J2EE [25] documentation for more detail.

[2] This transition occurred with introduction of local interfaces and container managed relations in the EJB specification version 2.0.

[3] J2EE specification version 1.3 was the latest stable specification at the moment this project was started.

Messaging destinations, such as topics and queues, are resources provided by a messaging service implementation. Data sources are resources provided by the application server for data access by business components into the enterprise information services (data) tier, and most commonly are exemplified by JDBC connection pools managed by the application server.

A J2EE programmer explicitly programs only EJBs and Web components. These custom-written components interact with each other and system services both implicitly and explicitly. For example, an EJB developer may choose explicit transaction demarcation (i.e., Bean-Managed Transactions) which means that the developer assumes the burden of writing explicit programmatic interaction with the platform's *TransactionManager* service through well-defined interfaces. Alternatively, the developer may choose Container-Managed transaction demarcation, where transactional behavior of a component is defined through its descriptors and handled completely by the EJB container, thus acting as an implicit dependency of the EJB on the underlying *TransactionManager* service.

2.3 Links Between Components

2.3.1 Remote Interactions

J2EE defines only three basic inter-component connection types that can cross application server boundaries, in all three cases, communication is accomplished through special Java objects.

- *Remote EJB invocation :* synchronous EJB invocations through EJB Home and EJB Object interfaces.
- *Java Connector outbound connection :* synchronous message receipt, synchronous and asynchronouse message sending, database query using ConnectionFactory and Connection interfaces.
- *Java Connector inbound connection :* asynchronous message delivery into Message-Driven Beans (MDBs) only, utilizing ActivationSpec objects.

In the first two cases, an application component developer writes the code that performs lookup of these objects in the component's run-time JNDI context as well as code that issues method invocations or sends and receives messages to and from the remote component. The component's run-time JNDI context is created for each deployment of the component. Bindings in the context are initialized at component deployment time by the deployer (usually by means of component's deployment descriptors). These bindings are assumed to be static, since the specification does not provide any contract between the container and the component to inform of any binding changes.

In the case of Java Connector inbound communication, ActivationSpec object lookup and all subsequent interactions with it are done implicitly by the MDB container. The protocol for lookup has not been standardized, though it is reasonable to assume a JMX- or JNDI-based lookup.

Assuming the underlying application server provides facilities to control each step of deployment process, establishment of a link between J2EE components would involve:

- deployment of target component classes (optional for some components, like destinations),
- creation of a special Java object to be used as a target component's proxy,
- binding of this object with component's host naming service (JNDI or JMX),
- start of the target component,
- deployment of referencing component classes,
- creation and population of referencing component's run-time context in its host naming service,
- start of the referencing component.

However, none of modern application servers allow detailed control of the deployment process for all component types beyond what is possible by limited options in their deployment descriptors [4]. Therefore our infrastructure will use a simplified approach that relies on features currently available on most application servers:

[4]For example, creation of EJB Home objects is usually automatically handled by the container, as well as its binding into JNDI. Some servers, notably JBoss, allow creating custom multiple EJB Home objects (utilizing different remote invocation transport protocols) for a single EJB deployment, however their deployment is still coupled with deployment of the component itself. Ideally, one should be able to deploy the EJB component and then dynamically deploy any number of transport-specific EJB Home objects.

- ability to deploy messaging destinations and data sources dynamically,
- ability to create and bind into JNDI special objects to access messaging destinations and data sources,
- ability to specify initial binding of EJB Home objects upon EJB component deployment,
- ability to specify a JNDI *reference* [5] in the referencing component's run-time context to point to the EJB Home binding of the referenced EJB component.

In our infrastructure which is limited to homogeneous application servers, these options are sufficient to control inter-component links through simple deployment descriptor manipulation. However, in context of heterogeneous application servers, simple JNDI references and thus simple descriptor manipulation are insufficient due to cross-application-server classloading issues.

2.3.2 Local Interactions

Some interactions between components can occur only between components co-located in the same application server JVM and sometimes only in the same container. In the Web tier, examples of such interactions are servlet-to-servlet request forwarding. In the EJB tier, such interactions are CMP Entity relations and invocations via EJB local interfaces. Such local deployment concerns need not be exposed at the level of a distributed deployment infrastructure other than to ensure colocation. Therefore, the infrastructure treats all components requiring colocation as a single component.

2.4 Deployment of J2EE Applications and System Services

2.4.1 Deployment of Application Components

Deployment and undeployment of standard J2EE components has not yet been standardized (see JSR 88 [10] for standardization effort [6]). Therefore, each application server vendor provides proprietary facilities for component deployment and undeployment. And while the J2EE specification does define packaging of standard components which includes format and location of XML-based deployment descriptors within the package, this package is not required to be deployable by an application server without proprietary transformation. Examples of such transformation are

- generation of additional proprietary descriptors that supplement or replace the standard ones,
- code generation of application server-specific classes.

In order to proceed with building a dynamic distributed deployment infrastructure capable of deploying in heterogeneous networks, we propose a universal unit of deployment to be a single XML-based deployment descriptor or a set of such, bundled into an archive. The archive may optionally include Java classes that implement the component and any other resources that the component may need. Alternatively, the deployment descriptors may simply have URL references to codebases.

We assume presence of a dynamic deployment/undeployment service on all compliant J2EE servers and a robust application server classloading architecture capable of repeated deployment cycles without undesired classloading-related issues. Most modern application servers (e.g., JBoss [11] and Geronimo [1]) do provide such facilities.

2.4.2 Deployment of System Components (Services)

While lacking only in the area of defining a clear specification of deployment and undeployment when it comes to application components, the J2EE standard falls much shorter with respect to system services. Not only a standardized deployment facility for system services is not specified, the specification, in fact, places no requirements even on life cycle properties of these services, nor does it address the issue of explicit specification of application component dependencies on the underlying system services. Instead it defines a role of human deployer who is responsible for ensuring that the required services are running based on his/her understanding of dependencies of application components on system services as implied by the nature of components and their deployment descriptors.

[5] A JNDI *reference* is a binding that automatically redirects any lookups to another JNDI binding, possibly on a different JNDI server. Please, refer to JNDI documentation [28] for more details.

[6] JSR 88 later became J2EE Application Component Deployment Specification and was introduced in J2EE specification version 1.4.

For example, an EJB with container managed transactions that declares at least one method that supports/requires/starts a new transaction would require presence of a *TransactionManager* service in the application server. Similarly, a message-driven bean implicitly requires an instance of a messaging service running somewhere in the network that hosts the messaging destination for the MDB and a Java Connector based hook-up from within its hosting application server to this messaging service.

Given that applications would typically use only a subset of services provided by an application server, componentized application servers that allow incremental service deployments depending on the needs of the application allow for most efficient utilization of server resources. There are several J2EE application servers that are already fully or partially componentized, including open source application servers JBoss [11] and JOnAS [19]. We feel that dynamic reconfiguration of application servers through dynamic deployment and undeployment of system services is essential to building a resource-efficient framework for dynamic distributed deployment of J2EE applications. Therefore we advocate and will use as a model a micro-kernel application server design used by the JBoss application server [8]. In this model a minimal server consists of a service invocation bus, a robust classloading subsystem, some naming subsystem and a dynamic deployment subsystem. All other services are hot-deployable and communicate through a common invocation bus. For example, JBoss utilizes a Java Management Extensions (JMX) [27] server that provides basic naming and invocation facilities. In addition JBoss implements an advanced classloading subsystem and a deployment service. All other JBoss services are dynamically deployable and expose themselves as JMX MBeans with well defined (un-)deployment mechanism and lifecycle. Such an application server design facilitates explicit handling of application component dependencies on system services and proper configuration and deployment of only required system services.

3 Infrastructure Architecture

3.1 Useful Definitions

Application server node, network node, target node or simply node is a computer system or a cluster of computers that run an instance of the infrastructure-controlled application server.

Application path is an abstraction that represents a deployment (potential or actual) of application component replicas on infrastructure nodes such that these replicas are configured to properly communicate with each other preserving original application semantics even in presence of other application paths for the same application.

Deployment specification is a description of a application paths as used by the infrastructure. It is written in the infrastructure-defined language and can be written manually, constructed by a planning algorithm or generated from a visual representation of an application path using special visual editors.

Component replica is a deployment of a component. There could be multiple deployments of the same component on different nodes and with different configurations. However, multiple replicas of stateful components may require state consistency management between replicas.

Link is an abstraction of connectivity between two components.

Link replica is an instance of a link used to connect specific ports of specific component replicas in a deployment specification.

3.2 Overview

The infrastructure consists of a network containing multiple application server nodes. Each application server node runs an infrastructure-controlled *Agent Service*. These agents communicate with an instance of a *Replication Management Service* (consisting of *Component Registry*, *Replica Configuration* and *Replica Deployment Services*) running on one network node (which can be dedicated). In principle, these services can be replicated to allow the infrastructure to scale, however, the current version of the infrastructure has not focused on the associated consistency issues. In addition, a *Deployment Unit Factory Service* (one or many) runs on some subset of the nodes (see Fig. 2).

The infrastructure defines architecture description languages (ADL) for component and link description and assembly. Main features of the *Component Description Language* are (1) a clear separation of system components from application

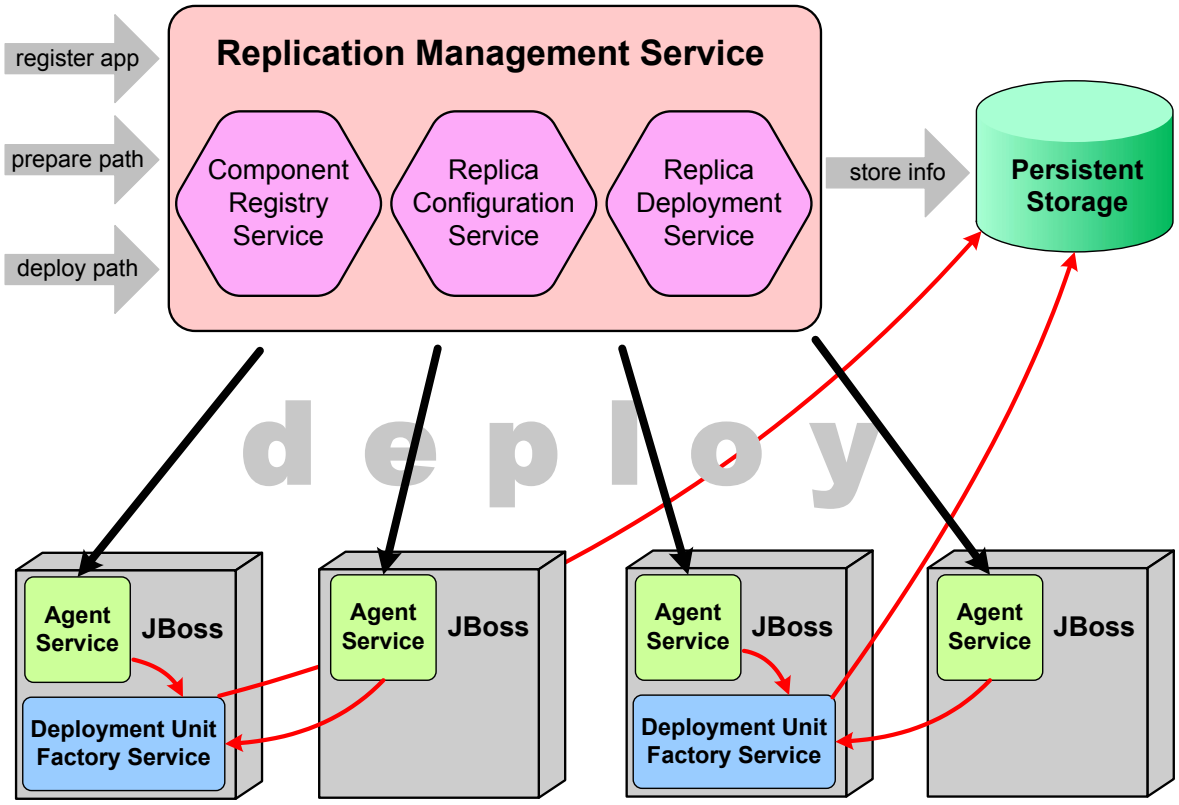

Figure 2: Infrastructure architecture.

components, (2) a flexible type system for component ports and links, (3) the ability to specify dependencies of both application and system components on other system components, and (4) a declaration and expression language for configurable component properties. The *Component (Replica) Assembly Language* allows for assembly of replicas of previously defined components into application paths by connecting appropriate ports via link replicas and specifying the mapping of these component replicas onto target application server nodes.

3.3 Infrastructure Usage

The usage of the infrastructure can be divided in the following set of steps (see also Fig. 2):

1. **Initialization.** The infrastructure is initialized with a description of available network nodes. This description is supplied by a network administrator, alternatively, the nodes may be configured to register themselves with the infrastructure and provide sufficient information about themselves.

2. **System components and application registration.** The infrastructure has to be initialized with descriptions of system and application components as well as links prior to any requests for deployment of replicas of these components. Multiple applications can be registered with the infrastructure at any time, as long as they have unique names. These descriptions (written in the Component Description Language) are registered with the *Component Registry Service*. It is expected that an application server provider prepares and registers a description of system services (system components) and links that are available for dynamic deployment on compatible target nodes, while the application vendor prepares a description of application components.

3. **Writing Deployment Path Specification.** The application deployer writes a deployment path specification in the *Component Assembly Language*. In it s/he specifies the placement of components (system and application) on the target nodes and links that connect them. The deployer may choose to write the specification by hand, or to use the

GUI path editing tool, which also serves as a user-friendly portal to the Replication Management Service. Figure 3 shows the GUI tool with a two-host distributed deployment of Java PetStore [23], one of the sample J2EE applications used in our experiments.

4. **Preparing deployment path.** After the initial registration, the infrastructure is ready to accept deployment requests. First a deployment specification for an application path is submitted for *preparation* to the *Replication Management Service*. It performs initial validation and passes the deployment specification to the *Replica Configuration Service*.

   The Replica Configuration Service, in turn attempts resolution of application component dependencies on system components and recursively, dependencies of newly discovered system components on other system components. If all component dependencies successfully resolve, the Configuration Service then configures each component replica. During configuration, the Configuration Service attempts to match any previously deployed replicas to replicas in the new path based on their configurations.

   All new replica deployment configurations are then persistently stored and any matched replicas that exist in other deployments are reused. This last step is sometimes called *committing prepared path*. Both success or failure of preparing the path are communicated back to the infrastructure user.

5. **Deployment of prepared path.** If path preparation and committing succeeded, the infrastructure client can subsequently request deployment of the prepared path. Upon a deployment request from the user, the *Replica Deployment Service* issues deployment requests to appropriate agents on nodes involved in providing services for this path. These agents, in turn, request deployable bundles of component replicas scheduled for deployment from a *Deployment Unit Factory Service*, located on a nearby node.

   For each requested component replica's deployment bundle, the Deployment Unit Factory service locates the corresponding replica configuration in the persistent storage and generates a properly configured deployment bundle. This bundle is then shipped to the requesting agent. The agent, upon receiving all deployable bundles for components and services scheduled for deployment on its node, deploys them in an order that respects deployment dependencies.

6. **Management of deployed paths.** The infrastructure maintains a registry of prepared paths, deployed paths and current state of application and system component replica deployments. Clients may request undeployment of previously deployed paths which will result in undeployment of component replicas that are exclusively used by the undeployed path.

4 Infrastructure Internals

4.1 Component Description Language

The primary goal of the *Component Description Language* is to describe components and links. The components (both system and application) and links are grouped into applications. An application defines a namespace for components and links that it contains. Application names must be globally unique. Components and links are differentiated by names that must be unique within the application. Absolute names of components are obtained by combining the application name with the component name relative to its containing application.

The Component Description Language differentiates between links, system components and application components. Application components are typically custom-developed for a given application, like web-tier and business-tier components (e. g. servlets, JSPs and EJBs). System components are typically services or resources that are part of the underlying application server, like JMS messaging service, transaction manager service, database service, and thus are usually shared by several applications running in the same application service. Certain resources that are technically provided to the application by system services are treated as application components, since they tend to represent resources that are in exclusive use by the application, for example, JMS messaging destinations, data sources to databases, etc.

4.1.1 Ports

The most significant difference between the application components and system components from the point of view of the Component Description Language is that application components declare *ports*, while system components do not. Ports of application components fall into two categories: *required* or *implemented*. Declaration of a required port in a component

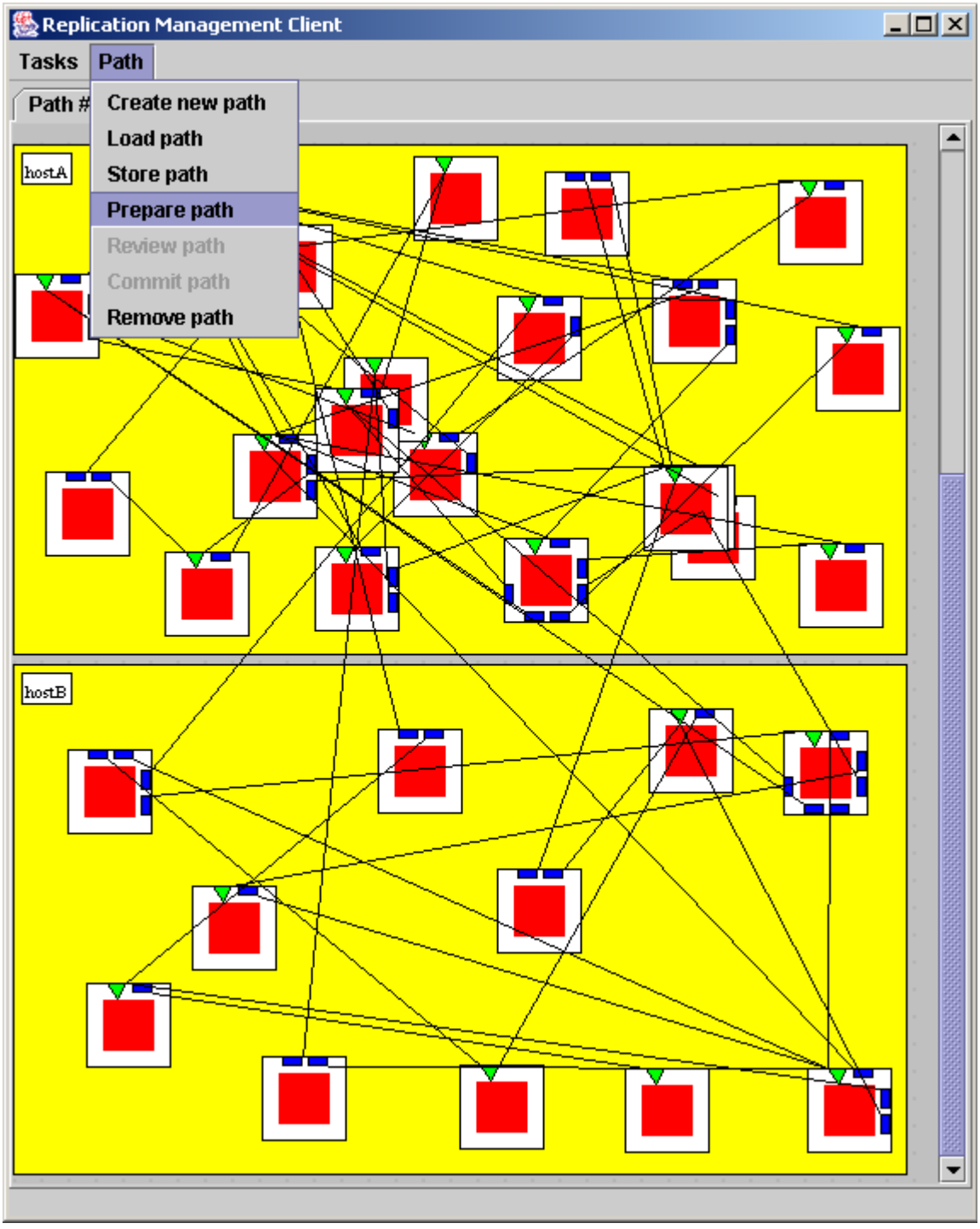

Figure 3: The GUI path editing and Replication Management Client tool with a distributed two-host deployment of the Java PetStore sample J2EE application.

description means that this component requires communication with another component. A declaration of an implemented port mean that the component can accept communication from another component (which in turn must have a matching required port). A port must have a unique name within each component description. It must also declare a *type* and a *link type*, these are used to check for semantic consistency of an assembly.

4.1.2 Port Type

Port type is a one of two mechanisms for assuring semantic consistency of component assembly. One can think of port types as interfaces to the component functionality. Port types are used in typechecking deployment specifications, so that a required port of one component replica with type P is connected to an implemented port of another component replica of type T only if T is a subtype of P. The infrastructure may use a pluggable type system and each application may define its own custom type system. The minimal requirements on a typesystem are that it implements checks for subtypes and exposes proper typesystem interfaces defined by the infrastructure.

4.1.3 Port Link Type

Link types specify what link may connect this port to another port. It is the second mechanism for assuring semantic consistency of a component assembly. The intuitive understanding of a link type is of a communication protocol through which functionality of a component may be accessed. Port link type's value must be a name of a well-defined link, known to the infrastructure. Typically there are only a few link types defined by an application server provider, corresponding to the three basic remote connectivity options available to components (see section 2.3.1).

4.1.4 Property Declaration Mechanism

Another feature of the Component Description Language is component and link *property declaration* mechanisms. It allows definition of adjustable component replica deployment configurations and at the same time expression of component dependency on system components. Property values are always strings, they can be null, a constant string, or a *property value expression* which evaluates to a string. Null property values are special cases, their function will be discussed later. Constant property values are simplest to understand, they remain constant for all replicas of the given component.

Properties of system components and links may be defined only in the component-wide or link-wide scopes. Application component properties may be defined in component-wide scope or in subscopes for each of the component's ports. The same way that an application acts as a namespace for its components, so components themselves act as namespaces for properties and ports, and ports act as namespaces for properties only. For example, if `myOnlineStore` application contains a component `StoreFrontEJB` that declares an implemented port `InvocationPort`, then

`myOnlineStore` corresponds to the full name of the application,

`myOnlineStore.StoreFrontEJB` is the full name of the `StoreFrontEJB` component,

`myOnlineStore.StoreFrontEJB.InvocationPort` is the full name of the `InvocationPort` implemented port of **`StoreFrontEJB`** component,

`systemId@myOnlineStore.StoreFrontEJB` is the full name of `systemId` property declared in the component-wide scope of the `StoreFrontEJB` component,

`JNDIBinding@myOnlineStore.StoreFrontEJB.InvocationPort` is the full name of `JNDIBinding` property declared in the `InvocationPort` subscope of the `StoreFrontEJB` component, which is really the subscope of the implemented port of `StoreFrontEJB`.

Values of component's properties are computed for each component replica and stored in a *replica configuration* during path preparation. More precisely, a *configuration* is a container for resolved property-value pairs from the corresponding property scope. Thus, a component replica will have a configuration corresponding to the component-wide property scope and a configuration for each of its ports. It can be queried for values of named properties and new property-value pairs can be added to it.

For application component replicas, configurations corresponding to component-wide scope and port scope are linked in a parent-child relationship for the purpose of property value query delegation. The delegation is from child to parent, so

that a failed lookup of a property in the port scope's configuration is delegated to the component-wide scope's configuration and fails only if the parent configuration also does not have a value for the given property. For example, in case of the `StoreFrontEJB` component, the `systemId` property is declared in the component-wide scope and its property-value is stored in the corresponding component-wide configuration and the configuration of the `InvocationPort` has no entry for the systemId property. However, a lookup of the `systemId` property against the configuration of the `InvocationPort` port, would succeed, because of the child-to-parent delegation. This is very similar to standard programming languages that allow nested scopes, where variables declared in the outer scopes are visible in the inner scopes.

A component replica's configuration (possibly, with nested subconfigurations for its ports), filled with resolved property values, completely defines this replica's deployment configuration. This configuration is subsequently used by the infrastructure to configure this replica's deployment descriptors.

4.1.5 Property Value Expression Language

The expression language for property values is very simple, it allows for concatenation of constant strings with values of other properties (of the same component replica or other components and links). Backus-Naur definition of the expression language is:

```
<CompositeExpression> ::= <SubExpression> |
                           <CompositeExpression><SubExpression>
<SubExpression>       ::= <string> |
                          ${<CompositeExpression>}|
                          ${<CompositeExpression>@<Namespace>}
<Namespace>           ::= <string>
```

The `${...}` operator is a ***value of*** operator, which performs value lookup of a named property that is specified inside the braces. Since all property values must be strings, the result of the lookup is a string that gets concatenated with preceding and following subexpressions. Property name inside the ***value of*** operator may be of the form `<name>@<namespace>`, where the `@` symbol separates the property name from the namespace in which the property is to be looked up. Alternatively, the namespace may be omitted, then the lookup will be done in the same namespace that contains the property whose expression is being evaluated.

4.1.6 Property Value Expression Example

Consider a component `myOnlineStore.StoreFrontEJB` that declares an implemented port `InvocationPort` and in that port, it declares a property `JNDIBinding`. Thus, this property's full name is `JNDIBinding@myOnlineStore.StoreFrontEJB.InvocationPort`. Let's say that the purpose of this property is to represent the JNDI name under which the EJBObject of the `StoreFrontEJB` EJB component replica is to be bound in its hosting node's JNDI.

```
<application name="myOnlineStore">
   <component name="StoreFrontEJB">
      <!--
         All components implicitly define
         systemId property.
      -->
      <implements>
         <port name="InvocationPort"
               type=...
               link-type=...>
            <property name="JNDIBinding">
               <!-- property value expression goes here -->
            </property>
         </port>
      </implements>
      ...
```

```
   </component>
</application>
```

Some possible value expressions for the `JNDIBinding` property and their explanations are:

`storeFrontEJB` : this is just a fixed string, the expression always evaluates to `storeFrontEJB`, which means that all replicas of this component will have the same value for this property. Since the value represents the name under which a replica binds its EJBObject, thus all replicas of this component would try binding their EJBObjects into JNDI using the same name, i.e. `/storeFrontEJB`. This should not create any problems if all replicas get deployed on different nodes and these nodes do not share JNDI namespaces. However, if two replicas end up on the same node and try to bind themselves to the same name, this would create a binding conflict.

`storeFrontEJB-${systemId}` : this value would evaluate to a string of form `storeFrontEJB-14`, if value of `systemId` property is `14`, thus allowing creation of unique values for JNDI bindings to avoid binding conflicts. Since lookup of `systemId` does not specify its namespaces, by default, `systemId` is looked up in the `myOnlineStore.StoreFrontEJB.InvocationPort` namespace which is the namespace to which `JNDIBinding` property belongs. Even though the `systemId` property is not declared in the `InvocationPort` scope but in the component-wide scope, the lookup should succeed, due to child-to-parent delegation.

4.1.7 Component Dependency Mechanism Through Property Value Expressions

In this subsection we shall describe the mechanism for expressing inter-component dependencies through property value expressions. As we have already described in section 4.1.5, value or a property may depend on values of properties from other namespaces. Such external references to properties in namespaces other than the component's own, is the mechanism by which a component expresses dependency on a system component. Consider the following example of an application component.

```
<component name="InventoryInvalidationTopic">
   ...
   <implements>
      <port name="DestinationPort"
            type="InventoryInvalidationTopic"
            link-type="jboss.system.jbossmq.DestinationLink">
          <property name="DestinationManagerObjName">
             ${DestinationManagerMBeanName@jboss.system.jbossmq.Service}
          </property>
      </port>
   </implements>
</component>
```

This descriptor snippet contains a declaration of property `DestinationManagerObjName@jps.InventoryInvalidationTopic.DestinationPort`. Expression value for this property is just a lookup of another property value, namely `DestinationManagerMBeanName@jboss.system.jbossmq.Service`, which means that their values must be the same. In this case, the referred namespace, `jboss.system.jbossmq.Service`, is in fact a name of a system component. This reference means that the `InventoryInvalidationTopic` requires a replica of `jboss.system.jbossmq.Service` component running on the same node and it must be fully configured before we can properly configure the `InventoryInvalidationTopic` component replica.

Last rule for SubExpression and definition of Namespace in the property value expression language (section 4.1.5) guarantees that referred namespaces can only be constants, therefore, what types of components a given component depends on may be determined statically[7] by analyzing the component's description. This also allows the infrastructure to perform additional component description validation through assertion of existence of component definitions of referred components.

If value expressions for several properties in the same scope (e.g., properties of one port) happen to refer to properties from the same external namespace, the infrastructure treats this as a dependency on the same replica of the component with name matching the external namespace.

[7] Note, however, that name of the property value which is being looked up in the external namespace can depend on values of other properties. This, in conjuction with custom property value evaluators provides a very expressive configuration and dependency mechanism.

4.1.8 Property Value Propagation Mechanism

In the previous subsection we introduced the mechanism for expressing dependencies on system components through property value expressions. In this subsection we discuss a closely related issue of property value propagation between components. Imagine a situation that a component (`Catalog`, in our example) has a required port that points to another component (`Item`). Link of type `jboss.system.EJBLink` (corresponding to the synchronous EJB invocation) connects these components. In order to properly connect to the `Item` component, the `Catalog` component needs to know some properties value from the `Item` component's namespace, for example the JNDI name of the `Item`'s Home Object.

We solve this problem by using certain mechanism of property value propagation between components, through properties of the link connecting the components. Let see how it works in the above example. The `Item` component specifies `EJBObjectJNDI` property of its implemented port:

```
<component name="Item">
   <implements>
      <port name="InvocationPort" type="Item"
                  link-type="jboss.system.EJBLink">
         <property name="EJBObjectJNDI">
            Item-${systemId}
         </property>
      </port>
   </implements>
   ...
</component>
```

The link type `jboss.system.EJBLink` has property `EJBObjectJNDI`, which, in the current example, evaluates in the context of the *target port*, that is, the implemented port of the `Item` component. The next XML snippet describes this property:

```
<link type="jboss.system.EJBLink">
    <property name="EJBObjectJNDI">
        ${EJBObjectJNDI@__targetPort}
    </property>
    ...
</link>
```

Now when the property value has been propagated to the link namespace, the last step in the chain is achieved by the following rule: if the property whose value is being evaluated is declared in the scope of a required port, and if the external namespace matches the link type of the port, it is then a reference to the link property:

```
<component name="Catalog">
   <requires>
      <port name="PortToItem" type="Item"
                  link-type="jboss.system.EJBLink">
         <property name="EJBObjectJNDI">
            ${EJBObjectJNDI@jboss.system.EJBLink}
         </property>
      </port>
      ...
   </requires>
   ...
</component>
```

4.2 Component Replica Assembly Language

The *Component (Replica) Assembly Language* is used for writing deployment (path) specification. This language allows for the deployer to request that a replica of a given application component be deployed on a particular node and how its ports are

connected to ports of replicas of other components within the deployment specification. Only application components can be assembled using the assembly language. This design choice is intentional in order to allow the application path planner to focus only on application aspects of the path without worrying about system components needed to support correctness of the application components' operation. It is the role of the infrastructure to resolve dependencies of the application components on system components, and subsequently to configure and deploy the required system components.

The deployer may choose to write the specification by hand, or to use the GUI deployment path editing tool. Figure 3 shows the GUI tool with a two-host distributed deployment of Java PetStore [23]. The following code snippet is an excerpt from the path specification produced by the GUI tool for the sample Java PetStore distributed deployment. It shows that the `Cart` component (with `replicaId` being 93) is deployed on the `hostB` and is connected by a link replica to the `Catalog` component (with `replicaId` being 4) residing on `hostA`.

```
<replication-path path-id="...">
   ...
   <component-replica replicaId="93">
      <configuration>
         <property key="name" value="jps.Cart"/>
         <property key="targetId" value="hostB"/>
         <port-configuration>
            <property key="name" value="jps.Cart.InvocationPort"/>
         </port-configuration>
         <port-configuration>
            <property key="linkType" value="jboss.system.EJBLink"/>
            <property key="name" value="jps.Cart.PortToCatalog"/>
         </port-configuration>
      </configuration>
   </component-replica>
   <component-replica replicaId="4">
   <configuration>
      <property key="name" value="jps.Catalog"/>
      <property key="targetId" value="hostA"/>
         <port-configuration>
            <property key="name" value="jps.Catalog.InvocationPort"/>
         </port-configuration>
         <port-configuration>
            <property key="linkType"
                      value="jboss.system.jca.DataSourceLink"/>
            <property key="name" value="jps.Catalog.PortToEstoreDS"/>
         </port-configuration>
      </configuration>
   </component-replica>
   <link-replica replicaId="157">
      <configuration>
         <property key="name" value="jboss.system.EJBLink"/>
         <property key="__destinationEndpoint_endpointId" value="4"/>
         <property key="__destinationEndpoint_endpointPortId"
                                    value="jps.Catalog.InvocationPort"/>
         <property key="__sourceEndpoint_endpointId" value="93"/>
         <property key="__sourceEndpoint_endpointPortId"
                                    value="jps.Cart.PortToCatalog"/>
      </configuration>
   </link-replica>
   ...
</replication-path>
```

4.3 Component Configuration Process

A deployment specification for an application path is a graph (with no cycles) of replicas of components connected via directed links from required to implemented ports (directed acyclic graph).

The component configuration process (preparing deployment path) is a *leaf-to-root, post-order* processing of the DAG. *Leaf* replicas are ones that have no required ports and thus have no outgoing links. However, they may depend on system components through the property value expression mechanism (section 4.1.7). So the algorithm in turn attempts resolution of application component dependencies on system components and recursively, dependencies of newly discovered system components on other system components. It then proceeds in the direction opposite to link direction. On this way, necessary property values are propagated from a component scope to the scope of its implemented port, then to the link, then to the required ports of the connected components, according to the property value propagation mechanism (section 4.1.8).

A replica in the graph is processed only after all component replicas that it connects to via its required ports are already processed. When configuring a replica, the following order of property resolution within component scopes is adopted: (1) component-wide scope, (2) implemented ports, (3) required ports. This means that component-wide scope will be filled with resolved property-value pairs first, and only after that all properties for all implemented and required ports are resolved. The order was chosen so that ports' properties may rely on component-wide properties being configured, so as to use their property values.

4.3.1 Component Reuse Algorithm

A component's replica on a given node can be safely reused by multiple deployment paths if the same sequence of communications with the replica will result in the same application state. Replica reuse is an obvious optimization that allows for decreased deployment overheads and consistency management. In the J2EE component model, a component is unaware of any components that require it and the specification disallows component reference cycles, so all paths are graphs without cycles. Moreover, in J2EE, any references to other components required by a given component must be set at this component's deploy time, so a component is configurable only at deploy time. The infrastructure adopts the following component reuse algorithm, which is performed as a part of the preparing of an application deployment path, after component dependencies have been resolved and all components' properties have been evaluated.

Imagine a component C with required ports labeled from 1 to n. A replica $R1$ of component C deployed on node N can be reused for a requested deployment of replica $R2$ of C on node N, only if component replicas $RQ1$, ..., RQn required by $R1$ can be reused for corresponding component replicas required by $R2$. This would imply that if a component has no required ports, then all replicas of this component deployed on the same node, would map to a single replica. This however poses problems because a component's semantic role in the application may depend not only on components it references but on references to entities not modeled as components by the infrastructure. For example, a data source application component has no required ports, however, it has properties that define database connection parameters that this data source component uses to connect to the external RDBMS. So, even though two data sources for the same logical database may be deployed on the same node, however the two datasources may be configured to communicate with two different RDBMSs (for example, one with test data and one with production data). In order to handle such cases, we introduce the notion of *primary component property*. Primary properties are usually the properties that need to be explicitly specified by the application deployer, such as the database host name for a data source application component. So, an additional reusability condition is added: a component replica $R1$ can be reused for replica $R2$ only if values of their primary properties are equal.

In fact, component *name* and *target host id* can be treated as two implicit primary properties of any component replica. So, the generalized reusability rule becomes:

> Component replica $R1$ can be reused in place of component replica $R2$ only if primary property values of these replicas are the same and subgraph of $R1$'s referenced component replicas can be reused in place of $R2$'s corresponding subgraph.

Primary property values together with hashcodes of the components that this component connects to through its required ports fully define unique id of the component.

5 Implementation

We implement the infrastructure as a part of the JBoss open source Java application server [11] and test it on several sample J2EE applications – Java PetStore [23], RUBiS [20] and TPC-W-NYU [32]. The infrastructure implementation utilizes the

JBoss's extendable *micro-kernel* architecture, based on the JMX [27] specification.

All infrastructure nodes run an instance of JBoss application server. These JBoss instances are configured to start a custom *Agent MBean*, which serves as the infrastructure-controlled Agent Service (section 3, see also Fig. 2). The Agent MBean plugs into JBoss deployment mechanism.

One master node runs a JBoss instance with the *XmlBlaster Service*, which acts as a persistence back-end, to store the information of prepared application paths, deployed paths and current state of application and system component replica deployments. XmlBlaster [34] is a Publish/Subscribe and Point-To-Point (PTP) Message-Oriented middleware (MOM) server, which exchanges messages between publishers and subscribers. Messages are described with XML-encoded meta information. A lot of features are supported, among them is a persistence support for messages. It is also equipped with the full text search capabilities – subscribers can use *regular expressions* or XPath [33] to filter the messages they wish to receive. Our persistence and inter-node messaging is accomplished through XmlBlaster.

Any number of infrastructure nodes may serve as hosts to codebase and deployment generation services. This functionality is encapsulated in a deployable Web application (J2EE WAR) – *Deployer* – serving as the Deployment Unit Factory Service. It contains the codebases of the applications preregistered with the infrastructure, in the JAR format. An Agent MBean requests the predefined Deployer for the deployable bundles for components and JBoss services scheduled for deployment on its node. The Deployer queries the XmlBlaster storage back-end for the replica configurations, produces deployable bundles and returns them back to the Agent, which in turn deploys them on the node in the order preserving component dependencies.

Often, replica configuration data obtained by the Deployer from the XmlBlaster backend is much smaller than the size of a deployment bundle, which may include classfiles and other content. Therefore it is resonable to install the Deployer service, which contains bulky codebases, close to network edges, so that it needs to obtain only small configuration data from the (centralized) XmlBlaster back-end. Ideally, it should be the responsibility of the Agent to determine which Deployer is closer to it.

As a part of the infrastructure, we have implemented a GUI tool serving as a Replication Management Service client. With this tool, infrastructure users may:

- compose and edit application deployment path specifications for preregistered applications, rather than writing them manually using the Component Replica Assembly Language;
- interact with the Replication Management Service for preparing, committing, deploying and removing application deployment paths.

Figure 3 shows the GUI tool with a two-host distributed deployment of the Java PetStore [23] sample J2EE application.

The Replication Management Service should run as a JBoss service or as a stand-alone application. In the current implementation, it runs as an application bundled with the GUI tool, with the *Java Event Notification* as the messaging mechanism between them. However, their codebases are decoupled and all necessary support for other pluggable (remote) communication mechanisms is available. Our future plans include implementation of a JMX-based communication mechanisms between the Replication Management Service and its clients (e.g., the GUI tool).

6 Related Work

The *deployment* and *dynamic reconfiguration* of (distributed) applications has been the subject of extensive research in the software engineering and distributed systems communities. In recent years increased attention was drawn to the deployment and reconfiguration problems of component-based applications, as the emerged component frameworks such as CORBA Component Model [18] and J2EE [25] matured. Research efforts in this direction can be vaguely divided into two camps. The first [2, 13] try to construct a general model for a relatively broad class of systems, by identifying the required functionality for dynamic reconfiguration, but rarely provide immediately applicable mechanisms for actual reconfiguration. The second [22, 3, 15] provide practical mechanisms for carrying out certain kinds of reconfigurations, but usually assume a specialized system architecture to do so. The work presented in this paper belongs to the second category.

It was acknowledged that *component dependencies* represent an important aspect of component-based systems, from the fault-tolerance, performance, management and reconfiguration perspectives. In [13] authors present a generic model of reifying dependencies in component systems. They identify two distinct kinds of dependencies: (1) requirements for loading a component into the system (called *prerequisites*), and (2) *dynamic dependencies* between loaded components in a running system. *Component prerequisites* are further subdivided into the three categories: (a) the *nature* of the hardware resources the component needs, (b) the *capacity* of the hardware resources it needs, and (c) the software services (such as

other components) it *requires*. QoS specification languages and architecture description languages (ADL) are proposed to express such specifications. To manage component's *dynamic dependencies*, the authors introduce the notion of *component configurator*, which is instantiated for each component and is responsible for storing the runtime dependencies between the component and other system and application components. The component configurator for component C keeps track of so-called *hooked* components, i.e., components on which C depends, and *client* components, which themselves depend on C. A flexible reconfiguration interface of component configurator allows for efficient management of component dependencies.

With regards to the generic dependency classification of [13], J2EE component model we are working with has only static dependencies (a.k.a. prerequisites), which come as a specification of system and application components that are required for a given component to execute. J2EE does not allow for dynamic reconfiguration of deployed components, so technically, there is no need for a dynamic component configurator, and J2EE deployment descriptors are sufficient for describing static deploy-time dependencies. In this work we do not address component hardware and QoS requirements at all, partly because it lies beyond the scope of the J2EE specification. Also in the J2EE component model, application components that provide services are typically unaware of their clients. The rationale for this awareness would be the ability to reconfigure clients in case the server component has to be taken off-line or reconfigured, but since J2EE does not allow for dynamic reconfiguration, client-awareness is not supported in J2EE.

Augmenting middleware with additional services that simplify the tasks performed by application developers, deployers and system administrators naturally follows the spirit of the middleware paradigm. Several previous studies proposed mechanisms of dynamic application reconfiguration through component redeployment and implemented them as middleware services. The work in [5, 6] proposed active monitoring and micro-reboots for fast automatic recovery and fault isolation. Authors of [7] advocate the approach of running multiple versions of the component at the same time, to reliably upgrade the system. The authors of [22] built a middleware service for atomic redeployment of EJB components across multiple servers. Our work follows this path, by proposing an infrastructure that facilitates and automates dynamic component deployment in distributed environments. However, this paper is different from the previous work in dynamic deployment and reconfiguration of component-based applications in that it specifically addresses the problem of efficiently expressing dependencies of portable J2EE application components and connectors on services provided by the middleware. We are working strictly within the constraints imposed by the J2EE programming model and do not propose extensions to the J2EE specification.

The variety of deployed components resulting from the usage of our infrastructure represents an *application-level overlay network* of J2EE components analogous to that of [2], [7], and [14], where several instances of the same component may coexist together. We believe that J2EE limitations on component lifecycle, concurrency and state may allow for efficient models of consistency between multiple versions of the same stateful J2EE component. The proposed infrastructure may form a foundation for a tool for J2EE *component replication*, analogous to the replication of CORBA components [16]. Replication of J2EE components can be used for different purposes, ranging from failover and increased availability to differentiation of the service among several client groups.

Acknowledgments

This research was sponsored by DARPA agreements N66001-00-1-8920 and N66001-01-1-8929; by NSF grants CAREER:CCR-9876128, CCR-9988176, and CCR-0312956; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Labs, SPAWAR SYSCEN, or the U.S. Government.

References

[1] Apache Software Foundation. Apache Geronimo Application Server. `http://geronimo.apache.org/`.

[2] N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. In *Proceedings of the 15th International Conference on Tools with Artificial Intelligence (ICTAI'03)*, November 2003.

[3] T. Batista and N. Rodriguez. Dynamic reconfiguration of component-based applications. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, June 2000.

[4] BEA Systems Inc. WebLogic Application Server. http://www.beasys.com/products/weblogic/.

[5] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. JAGR: An autonomous self-recovering application server. In *Proceedings of the 5th International Workshop on Active Middleware Services*, June 2003.

[6] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, Internet services. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2002.

[7] J. E. Cook and J. A. Dage. Highly reliable upgrading of components. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, May 1999.

[8] M. Fleury and F. Reverbel. The JBoss extensible server. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware'2003)*, June 2003.

[9] IBM Corporation. WebSphere Application Server. http://www.ibm.com/software/websphere/.

[10] Java Community Process. *Java Specification Request 88 (JSR88).* http://www.jcp.org/en/jsr/detail?id=88.

[11] JBoss Group. JBoss Application Server. http://www.jboss.org.

[12] T. Kichkaylo, A. Ivan, and V. Karamcheti. Constrained component deployment in wide-area networks using AI planning techniques. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, April 2003.

[13] F. Kon and R. H. Campbell. Dependence management in component-based distributed systems. *IEEE Concurrency*, 8(1):26–36, 2000.

[14] D. Llambiri, A. Totok, and V. Karamcheti. Efficiently distributing component-based applications across wide-area environments. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 412–421, May 2003.

[15] J. Magee, A. Tseng, and J. Kramer. Composing distributed objects in CORBA. In *Proceedings of the Third International Symposium on Autonomous Decentralized Systems (ISADS'97)*, pages 257–263, 1997.

[16] V. Marangozova and D. Hagimont. An infrastructure for CORBA component replication. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 257–263, 2002.

[17] Microsoft Corporation. *Microsoft .NET.* http://www.microsoft.com/net/.

[18] Object Management Group. *CORBA Component Model (CCM) Specification.* http://www.omg.org/technology/documents/formal/components.htm.

[19] ObjectWeb Consortium. JOnAS Application Server. http://jonas.objectweb.org/.

[20] ObjectWeb Consortium. RUBiS: Rice University Bidding System. http://rubis.objectweb.org/.

[21] Oracle Corporation. Oracle Application Server. http://www.oracle.com/appserver/.

[22] M. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, and A. L. Wolf. Reconfiguration in the Enterprise JavaBean component model. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 67–81, 2002.

[23] Sun Microsystems Inc. Java Pet Store Sample Application. http://java.sun.com/developer/releases/petstore/.

[24] Sun Microsystems Inc. *Enterprise JavaBeans (EJB) Specification.* http://java.sun.com/products/ejb/.

[25] Sun Microsystems Inc. *Java 2 Enterprise Edition.* http://java.sun.com/j2ee/.

[26] Sun Microsystems Inc. *Java Connector Architecture (JCA) Specification.* http://java.sun.com/j2ee/connector/.

[27] Sun Microsystems Inc. *Java Management Extensions (JMX) Specification.* http://java.sun.com/products/JavaManagement/.

[28] Sun Microsystems Inc. *Java Naming and Directory Interface (JNDI) Specification.* http://java.sun.com/products/jndi/.

[29] Sun Microsystems Inc. *Java Server Pages (JSP) Specification.* http://java.sun.com/products/jsp/.

[30] Sun Microsystems Inc. *Java Servlets Specification.* http://java.sun.com/products/servlet/.

[31] C. Szyperski. *Component Software*. Addison-Wesley, November 2002.

[32] TPC-W-NYU. A J2EE implementation of the TPC-W benchmark. http://cs1.cs.nyu.edu/~totok/professional/software/tpcw/tpcw.html.

[33] World Wide Web Consortium. *XML Path Language (XPath) Specification.* http://www.w3.org/TR/xpath.

[34] XmlBlaster Open Source Project. http://www.xmlblaster.org/.