Reconfiguration in the Enterprise JavaBean Component Model

Matthew J. Rutherford, Kenneth Anderson, Antonio Carzaniga,
Dennis Heimbigner, and Alexander L. Wolf

Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430 USA
{matthew.rutherford,kena,carzanig,dennis,alw}@cs.colorado.edu

Abstract. Reconfiguration is the process of applying planned changes to the communication, interconnection, componentization, or functionality of a deployed system. It is a powerful tool for achieving a variety of desirable properties of large-scale, distributed systems, including evolvability, adaptability, survivability, and continuous availability. Current approaches to reconfiguration are inadequate: some allow one to describe a system's range of configurations for a relatively broad class of system architectures, but do not provide a mechanism for actually carrying out a reconfiguration; others provide a mechanism for carrying out certain kinds of limited reconfigurations, but assume a specialized system architecture in order to do so. This paper describes our attempt at devising a reconfiguration mechanism for use with the popular and widely available Enterprise JavaBean (EJB) component container model. We describe extensions to the basic services provided by EJB to support the mechanism, a prototype implementation, and a case study of its application to a representative component-based distributed system.

1 Introduction

Subsequent to their development, software systems undergo a rich and complex set of management activities referred to as the *deployment life cycle* [3, 6]. These activities include the following.

- *Release:* packaging all artifacts and configuration descriptions needed to install a system on a variety of platforms.
- *Install:* configuring and assembling all artifacts necessary to use a released system. Typically this involves selecting from the release the configuration that is compatible with the specifics of the intended operating environment.
- *Activate:* putting installed software into a state that allows it to be used. Typically this involves allocating resources.
- *Deactivate:* putting installed software into a state that makes it unable to be used. Typically this involves deallocating resources.

- *Reconfigure:* modifying an installed and possibly activated system by selecting a different configuration from an existing release. Typically this activity is intended to satisfy an anticipated variation in operational requirements and, thus, is driven by external pressures.
- *Adapt:* modifying an installed and possibly activated system by selecting a different configuration from an existing release. This activity differs from *reconfigure* in that it is intended to maintain the integrity of the system in the face of changes to the operating environment and, thus, is driven by internal pressures.
- *Update:* modifying an installed and possibly activated system by installing and possibly activating a newly released configuration.
- *Remove:* removing from the operating environment the installed artifacts of a system.
- *Retire:* making a release unavailable for deployment.

Many commercial tools exist to address the "easy" part of the problem, namely the activities of release, install, remove, and retire (e.g., Castanet, InstallSheild [7], netDeploy [12], and RPM [1]), but none that covers all the activities. Research prototypes have made strides at addressing dynamic reconfiguration, but are generally conceived within restricted or specially structured architectures [2, 8, 9, 11].

In this paper we present our attempt at improving support for the activities of reconfigure, adapt, and update. Although the context and drivers for these three activities differ substantially, they clearly share many of the same technical challenges with respect to the correct and timely modification of a system. Therefore, in this paper we simply refer to the three activities collectively as "reconfiguration", using the individual activity names only when necessary.

In earlier work, we developed a tool called the Software Dock to automate the configuration and reconfiguration of distributed systems [3–5]. However, the Software Dock is currently restricted to the reconfiguration of installed systems that are not active. Activated systems complicate support for reconfiguration in at least three ways: (1) maintaining consistent application state between modifications; (2) coordinating modifications to concurrently active components; and (3) ensuring minimum disruption or "down time".

To help us better understand the challenges of reconfiguring activated systems, we embarked on an effort to study the problem in a particular context. The context we chose is the widely used Enterprise JavaBean (EJB) component framework [10]. EJBs are distributed components and, thus, further raise the level of complexity of software deployment activities, since the activities may have to be coordinated across multiple network nodes.

Further framing the problem, we delineated a space of reconfigurations that we wished to address in the study (Table 1). We consider three media of modifications leading to reconfiguration: parameters, implementations, and interfaces. We also consider whether or not modifications to multiple EJBs are dependent or independent; a dependency implies the need for transactional modification. In fact, the modifications should be both synchronized and transactional, since

the system could be unstable if the reconfiguration is not successful on all nodes. On the other hand, there may be modifications that do not change the contract between components, and while it may be desirable for these changes to be made on all nodes, the changes do not need to be coordinated.

	Independent	**Dependent**
Parametric	Preprogrammed modification applied to a single component.	Preprogrammed modification applied to multiple components.
Implementation	Modification to the implementation of a component that does not require a modification to the implementation of its clients.	Modification to the implementation of a component that requires a modification to the implementation of its clients.
Interface	Modification to the interface of a component that does not require a modification to its clients.	Modification to the interface of a component that requires a modification to its clients.

Table 1. Kinds of Reconfigurations.

A parametric reconfiguration is one that a component is itself designed to handle. It reflects a common way of planning for change in software systems, namely by having certain options specified as parameters through some external means such as a property file or database entry. By changing the parameter values, modifications can be made to the behavior of a software system without having to modify any of its executable software components. This type of reconfiguration might have to be coordinated across multiple nodes. For example, a parameter might be used to control whether or not a communication channel is encrypted, requiring distributed, communicating components to coordinate their response to a modification in this parameter.

An implementation reconfiguration is one in which only the implementation of a component is modified, but not its interface. Of course, the goal of separating implementation from interface is, in part, to isolate implementation modifications. Nevertheless, in some cases the effect of the modification does indeed propagate to the clients of the component. For example, a component may expose a method that takes a single string as its argument. The format of this string is important, and an implementation modification in the component may alter the expected format of this argument. This would require all client components to also modify their implementation to conform to the new format, even though the exposed interface did not change.

Finally, an interface reconfiguration results from the most pervasive modification to a component, affecting both the interface and implementation. Typically, an interface modification must be coordinated with client components. But this need not always be the case. For example, the modification may simply be an extension to the interface.

While this space may not be a complete expression of reconfiguration scenarios, it is sufficiently rich to exercise our ideas. In the next section we provide some background on the EJB framework, pointing out some of its shortcomings with respect to reconfiguration. Following that we introduce BARK, a prototype infrastructure for carrying out sophisticated EJB reconfigurations. We then demonstrate BARK by applying it to the reconfiguration of a distributed application that we built. The demonstration covers the six kinds of reconfigurations described above.

2 Background: Enterprise JavaBeans

The Sun Microsystems Enterprise Java initiative is a suite of technologies designed to provide a standard structure for developing component-based enterprise applications. The technologies address issues such as database connectivity, transaction support, naming and directory services, messaging services, and distribution. EJBs are the cornerstone of the Enterprise Java initiative. EJBs are distributed components, intended to execute within so-called *containers* that handle much of the complexity inherent in multi-threaded, database-driven, transactional applications; theoretically, the use of the EJB framework should allow developers to concentrate on the business logic of applications. The EJB specification provides strict guidelines about how EJB components must be packaged, and how they can reference other components. These guidelines provide a structure in which an automated deployment system can handle various management tasks.

EJBs come in three flavors: stateless session beans, stateful session beans and entity beans. The EJB 2.0 specification also defines a fourth flavor, message-driven beans, which are invoked by the arrival of a message to a specific topic or queue; here we only deal with the first three types of EJB. Stateless session beans are components that do not maintain any state between invocations. Essentially, stateless session beans provide utility functions to clients. Stateful session beans are components that need to maintain a conversational state on a per-client, per-session basis. That is, a different instance of a stateful session bean implementation class is used for each client, and its state is maintained between method invocations until the session is terminated. Entity beans are used to handle persistent business objects. This means that they are used to represent objects whose state can be shared across all the clients of the system. Typically, entity beans are used as a software representation of a single row of a query into a relational database.

Part of the EJB framework relates to the so-called "life cycle" stages that an EJB implementation instance goes through as it is servicing requests. Stateless session beans have a very simple life cycle, since the same instance can repeatedly service requests from different clients without the special handling that is required for stateful beans. For stateful beans, the life cycle is a bit more complicated, since the instance must be associated either with a particular client across multiple method calls or with a particular persistent entity. EJB

containers maintain the state of component implementation instances using the following four EJB life-cycle methods.

- *Activate:* the first method called after a stateful EJB is deserialized from secondary storage. Any resources that it needs should be allocated.
- *Passivate:* the last method called before a stateful EJB is serialized to secondary storage. Any resources that it holds should be released.
- *Load:* valid for entity beans only, this method instructs the instance to retrieve current values of its state from persistent storage.
- *Store:* valid for entity beans only, this method instructs the instance to save current values of its state to persistent storage.

Note that in EJB terminology, "deployment" is the process by which an EJB server loads an EJB package and passes it to the container to make it available to clients. In this paper we avoid this restricted definition of deployment, since it can be confused with the broader meaning described in Section 1.

Once the classes that comprise an EJB have been developed, they then must be packaged in a standard fashion so that EJB servers can install them properly. Typically, all the classes that are needed for the EJB to run (excluding system classes that are available to all components) are assembled into a JAR (Java Archive) file that includes deployment descriptors identifying the standard set of classes that permit the EJB to be managed and used (the so-called home, remote, and implementation classes). The descriptors also typically include the JNDI (Java Naming and Directory Interface) name to which the interfaces to these classes are bound. Another important part of the standard deployment description includes information about any other EJBs upon which the given component depends.

The packaging specification for EJBs allows multiple EJBs to be packaged together in a single EJB JAR file. An EJB container handles all of the EJBs packaged together as a single application unit; once EJBs are packaged together, they cannot be maintained separately from the other EJBs with which they were packaged. In our work we therefore assume that a single EJB JAR file is described as containing a single component, with each EJB in the package representing a different view onto that component. Thus, the decision made by the software producer about the packaging of EJBs essentially drives the granularity of how the deployment life cycle of those EJBs can be managed.

Deployment in EJB-based systems involves various combinations of a small set of common actions.

- *Retrieve* a new component package from a software producer (*install* and *update*).
- *Load* a component package into an EJB server (*activate*, *reconfigure*, *update*, and *adapt*).
- *Unload* a component package from an EJB server (*update*, *deactivate*, and *reconfigure*).
- *Reload* a component package into an EJB server to freshen its bindings to other components (*reconfigure*, *update*, and *adapt*).

– *Modify* a database schema, database data, or content file (*activate*, *update*, *reconfigure*, and *adapt*).

One of the major problems with these actions is that they can be heavy handed. This is especially true of the actions that must be taken to reconfigure, update, or adapt an activated system, where component packages must be reloaded just to make sure the bindings are up to date. This invasive process implies that some or even all of the components in a system must be shut down for some period of time, which is simply unacceptable in high-volume, high-availability applications such as the electronic-commerce infrastructure systems that EJBs were largely meant to support.

To a certain extent, this problem of heavy handedness is dictated by the way that component package descriptors are used to specify the dependencies among EJBs. Included in the package descriptor for a component is the well-known naming service name of all other referenced components. When a referenced component is loaded by its EJB application server, the well-known name is bound into the naming service and the component is available. This presents a problem when updating a component that has dependent clients: If the same well-known name is chosen for the new version of the component, then the existing version must first be unloaded before the new version can be loaded, meaning that the system will effectively be unable to satisfy requests for a time. If instead a different well-known name is chosen for the new version of the component, then the package descriptors for all of its clients must be updated to be told of this new name, which means that they must be reloaded by their EJB application servers, again resulting in down time.

3 Approach: The BARK Reconfiguration Tool

Our approach to the problem is embodied in a prototype tool we call BARK (the Bean Automatic Reconfiguration frameworK). It is designed to facilitate the management and automation of all the activities in the deployment life cycle for EJBs. BARK provides some basic functions, such as the ability to download component packages over the Internet and load them into the EJB container. Other, more sophisticated aspects of the framework manipulate the component package descriptors to provide fine-grained control over a system formed from EJBs, even down to the level of individual bindings between components. In a sense, the functionality that BARK presents to its users defines an "assembly language" for EJB deployment management.

It is important to note that BARK is only a management tool; it does not provide any analysis of the running system, nor does it make recommendations or determine automatically what steps need to be taken to reconfigure a system in a particular way. As a management tool, it provides a certain level of checking to make sure that the user does not put the system into an unstable state unwittingly. However, BARK generally allows the user to force any action, thereby allowing the user to fully control the reconfiguration of the system as they see fit.

3.1 Architecture

The high-level architecture of BARK is depicted in Figure 1 and consists of the following major components.

- *Application Server Module (ASM):* works in cooperation with an EJB application server instance to provide management of the components that it serves. The ASM is installed with every application server instance that is involved in the system being managed. The division of labor between an ASM and its application server is clear. The ASM is responsible for all the processing involved with managing the deployment life cycle of the components, including downloading packages, tracking deployment state and component versions, and managing reconfiguration transactions. In turn, the application server is responsible for handling the normal running of the software. An ASM only interacts with the application server directly when it activates or deactivates components. To achieve the necessary granularity for bindings, the ASM also adjusts the values stored in the JNDI naming service that is typically running as part of the application server.
- *Repository:* a central location for storing component software packages. ASMs download software packages from the repository when required. The repository is a conceptual part of the BARK system; there is no specialized repository software provided as part of BARK. In practice the repository can be any file server that is available to the ASMs. This represents the location to which a software producer would release software components.
- *Workbench:* provides a system administrator with a tool for directly controlling ASMs, serving as a bridge between the repository, which is controlled by software producers, and the ASMs, which represent the software consumers.

The bindings depicted in Figure 1 represent client/server relationships between components. Knowledge about such bindings are important for deployment activities, since it is changes in these bindings that are often involved in reconfiguration. The commands represent requests for deployment actions. Most of the commands depicted in Figure 1 are shown as originating from the workbench. However, inter-ASM commands are also necessary to ensure that any changes to bindings are approved by both the client and the server before they are completed.

BARK provides the ability to execute individual deployment actions or to execute scripts that combine multiple actions. Aside from conveniently grouping related sequences of actions, scripts are used to indicate transactional actions directed at one or more ASMs. In fact, a script is treated just as any other component and, therefore, its actions are managed by the ASM and the EJB application server. Scripts can also contain code for making changes to database structures or file contents.

3.2 Commands

Each command executed by an ASM is atomic and affects a single component or binding. Following are the primary ASM commands.

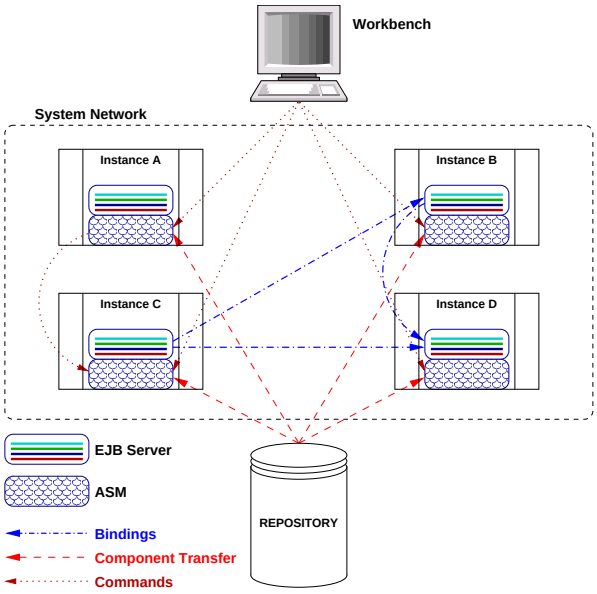

Fig. 1. The BARK Architecture.

- *Add:* directs an ASM to download to the local node a particular EJB component package from a repository. After the package is downloaded, the ASM processes its contents and makes changes to the package descriptor that will allow the component to be properly controlled by subsequent commands.
- *Delete:* directs an ASM to remove a package from the local node.
- *Load:* directs an ASM to trigger its local application server to enable a component to make bindings to other components and accept client requests. The implementation of this command is EJB-vendor specific, since each vendor has their own notion of how to "deploy" component packages.
- *Unload:* directs an ASM to disable a component from making bindings to other components and accept client requests. A component cannot be unloaded if it still has active bindings and/or bound clients.
- *Bind:* directs an ASM to make a client relationship to a server component. If the two components are managed by different ASMs, then the ASM first contacts the server ASM directly to make sure that both the client and server were correctly specified and can create the binding. After a component is fully bound, it is available to accept clients. In situations where there are circular dependencies among components, the *Bind* command can force a server to accept clients before it is fully bound.
- *Rebind:* directs an ASM to update the binding of a component. Unlike the *Bind* command, the client component must already be bound to a server component.

- *Unbind:* directs an ASM to remove a relationship between two components. This command normally cannot be used until all clients of a component are removed, but can force the removal of a relationship in cases of circular dependencies.
- *Execute:* directs an ASM to begin execution of a loaded and fully bound component.
- *Stop:* directs an ASM to suspend execution of an executing component.
- *Reload parameters:* directs an ASM to cause a component to reload its parameters.
- *Refresh bindings:* directs an ASM to cause a component to refresh its bindings.

The last two commands are specifically intended to reduce the down time suffered by a running application when undergoing reconfiguration. However, unlike the other commands, they are predicated on cooperation from component designers, who must expose interfaces explicitly supporting these two non-standard lifecycle activities. With conventional EJB application servers, the only way to cause a freshening of bindings or reloading of parameter values is to force the server to reload the entire component package. Either that or the component could be programmed to always refresh or reload before each use, just in case there was a change. Both these approaches exhibit performance problems. The alternative that we advocate through BARK is a compromise that allows a refresh or reload directive to come from outside the component when necessary to support reconfiguration.

3.3 Transactions

The programmatic interface of BARK is modeled through a class called `Connection`. Instances of `Connection` provide strongly typed methods for executing commands on the ASM with which it is associated. In order to get proper transactional processing, some care must be taken when using `Connection`. To illustrate this, the steps taken by the scripting engine within an ASM are described below.

1. *Retrieve `Connection` object from primary ASM instance.* If multiple ASMs are involved in the execution of a script, one of them needs to be chosen as the primary instance. This can be done arbitrarily, since there is no special processing that must be performed.
2. *Retrieve `Connection` objects for other ASMs.* The primary `Connection` object is used to open connections onto the other ASMs. These new, secondary `Connection` objects are all considered to be in the same transaction context as the primary `Connection` object.
3. *Issue commands.* With the proper connections established, commands can be issued to any of the ASMs.
4. *Commit or rollback.* Using the primary `Connection` object, the entire transaction can be committed or rolled back.

Thus, a key feature of BARK is its support for transactional reconfigurations, since it is easy to see how a partial reconfiguration could render a software system unusable. The requirements on BARK's transactions are as follows: all of the configuration changes must be committed or rolled back as a unit; none of the configuration changes can be visible to other transactions until they are committed; and configuration changes made to one component during the transaction must be visible to the other components that are involved in the transaction.

BARK uses an optimistic concurrency control scheme, which effectively means that the first transaction to modify an object will succeed, while subsequent modifications will fail on a concurrency violation error. When a transaction is committed through a primary `Connection` object, all the secondary `Connection` objects are committed using a standard two-phase commit protocol. Although application servers that support EJBs must provide a transaction manager that implements Sun's Java Transaction Service, BARK manages its own transactions. This was an implementation decision driven by the desire to keep ASMs cleanly separated from application servers.

3.4 Scripts

Scripts are XML descriptions defining sequences of BARK commands that should be carried out in a single transaction. The script first defines all of the ASMs that are involved in the script, and then provides the sequence of actions that must be performed on each of them.

Figure 2 contains a sample BARK script. In this script, only one node is being managed. The script first retrieves two component packages from the repository, giving them script-local names `compA` and `compB`. In steps 3 and 4, both components are loaded into the EJB application server. Step 5 binds `compB` to `compA`.

3.5 Name Binding

Most EJB application servers use JNDI as a mechanism to make the developer-defined name of the server's (home) interface available to clients. Clients of the component "know" this name, and use it to do their lookups. This arbitrary name poses some problems when maintaining components in a server. For one thing, the clients have this name hard coded somewhere in their software or in their deployment descriptors. This effectively means that the component must always be installed with this name, or clients will not know how to look up references to it. For another, JNDI only allows one object to be bound to a particular name, so new versions of a component force prior versions to be unloaded.

The primary mechanism that BARK uses to achieve finer control over component bindings, and thereby greater flexibility in changing those bindings, is to transform the JNDI bindings after a component has been loaded by a server. In essence, the idea is to create an internally unique name that is known to BARK. For example, BARK might generate the internal name `bark/7730183/12876309` for a component whose original JNDI name was `componentb/ComponentB`. This JNDI name rewriting is handled automatically.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE bark-script SYSTEM "bark-script.dtd" >
<bark-script name="SampleScript">
    <!-- host declarations -->
    <instance id="moleman" url="moleman:1099" />
    <!-- actions -->
    <get sequence="1" instance-id="moleman" id="compA"
         remote-url="http://repository/bark/ComponentA.jar" />
    <get sequence="2" instance-id="moleman" id="compB"
         remote-url="http://repository/bark/ComponentB.jar" />
    <load sequence="3" instance-id="moleman" component-id="compA" />
    <load sequence="4" instance-id="moleman" component-id="compB" />
    <bind sequence="5" instance-id="moleman" force="false"
         client-component-id="compB" client-view-name="ComponentB"
         server-component-id="compA" server-view-name="ComponentA" />
</bark-script>
```

Fig. 2. A BARK Script.

One drawback of this approach is that a client of a component managed by BARK, but that is not itself managed by BARK, does not have any way of binding to that component directly. To alleviate this problem, BARK provides an alias mechanism through which a name can remain unchanged while its association with internally generated names can change. The commands *Bind-name* and *Unbind-name* are provided to manipulate such aliases. For example, if there was a non-BARK application that needed to use the BARK-managed EJB `componentB`, and it was expecting its interface to be bound to the JNDI name `componentb/ComponentB`, then the *Bind-name* command could be used to automatically create an alias from `componentb/ComponentB` to the BARK-generated name `bark/7730183/12876309`.

3.6 Implementation

As mentioned above, the ASM software was designed to run in conjunction with an EJB application server. We use the open-source JBoss EJB application server in our prototype implementation of BARK. This server, as are other major servers such as BEA's WebLogic, is built using the Sun Microsystems JMX framework. JMX organizes services as so-called Manageable Beans (MBeans) that can be plugged together easily in a single application and so provide services to each other. The ASM software was integrated into the JBoss server as an MBean. This allows the ASM to have direct access to some of the important services that it needs, particularly the JNDI naming service and the EJB loading service. By being incorporated directly into the server application, the ASM is started and stopped when the server is, allowing for explicit control of the components it manages during those events.

4 Example Application: Dirsync

We developed an application called Dirsync as an exercise in the use of BARK. Dirsync provides a service for synchronizing the contents of shared directories on remote computers. It is a component-based distributed system built from a number of interdependent EJB components. Figure 3 shows the use relationships among the components; unfortunately, space does not permit an explanation of their individual functionality. A separate instance of Dirsync resides on each node in a network and is responsible for the directories on that node. The relationship between any two such instances, which is mediated by bound instances of `DataChannel` on either end, can be master/slave or peer-to-peer. Several versions of Dirsync were developed to drive a representative case study of how such a distributed component-based system can be evolved over time.

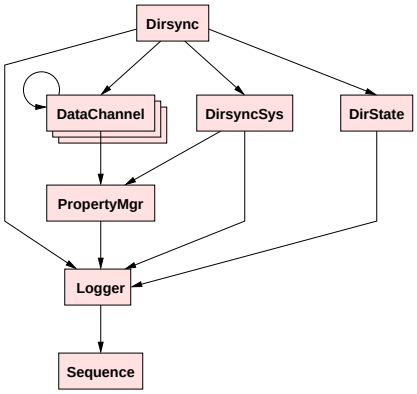

Fig. 3. The Dirsync Component Uses Hierarchy.

Most of the components that comprise the system are session beans, both stateless and stateful. A few entity beans are used to handle persistence of directory state, component parameters, and logging information. None of the entity beans are accessed directly by clients of a component. Instead, clients bind to a session bean that hides some of the details of the entity bean from the clients.

Although there are not a large number of components in Dirsync, complexity arises out of the relationships among them (both within a local machine and across the network), the changeable nature of the network topology, and the state that must be maintained on each local machine. These combine to make Dirsync a reasonable reconfiguration challenge. Some of the specific features of Dirsync that are representative of many real-world applications are as follows.

– *Both client/server and peer-to-peer relationships.* Client/server architectures are common, so any solution must be able to deal with them. But they are

also easier to reconfigure, since they embody a clear hierarchy and, therefore, it is usually straightforward to determine an ordering for changes. With peer-to-peer architectures, certain reconfigurations need to occur on multiple nodes at once, making the coordination of reconfiguration more difficult.
- *A non-stop service.* This is a very common characteristic of multi-user distributed systems: there must always be something running that is ready to accept requests. This presents a reconfiguration challenge because changes must be made with minimal disruption to the service.
- *Network topology is dictated by application settings.* A dynamic network topology is fairly common, and presents a reconfiguration challenge because inter-node dependencies can only be determined based on the current configuration. This also means that some changes to the application parameters will require new bindings to be made between remote components.
- *There is persistent state.* The use of a database, or some other persistence resource, is very common in distributed systems, particularly for business applications. This presents a reconfiguration challenge because changes must be coordinated between an external software application and the software components of the system.

Table 2 summarizes a sequence of reconfigurations that we applied to a simple scenario of Dirsync running on two nodes, `N-1` and `N-2`. Those particular reconfigurations were chosen because they represent a range of modifications commonly applied to component-based distributed systems and because they cover the space of reconfigurations outlined in Table 1 of Section 1.

We collected some initial performance statistics that indicate reasonable overhead in carrying out the reconfigurations. For example, the *Bind*, *Rebind*, and *Unbind* commands generally took on the order of 30 to 50 milliseconds. The *Add* and *Load* commands took much longer, consuming on the order of 1000 to 2500 milliseconds in our experiments. Our analysis shows that the *Add* command is dominated, not surprisingly, by network latency and repository implementation issues, while the *Load* command is dominated by time spent inside the JBoss server implementation of component activation. Clearly, more experiments are needed to validate these preliminary results.

5 Conclusion

The contribution of the work described in this paper lies primarily in the experience gained engineering advanced reconfiguration capabilities into an established component management framework. The challenge was to work within the confines of the services that the framework already provided. In fact, we saw the need to extend that framework in only two ways (reloading parameters and refreshing bindings), yet those extensions are really only for the purposes of reducing system down time and are not required functionality.

The next step for this work is to feed our experience back into the design of a next-generation Software Dock deployment system. Our intention is to create a version that is in some sense parametric with respect to the underlying

Release 1: Initial Deployment
Establishes the sharing of `dir1` in a client/server relationship between `N-1` and `N-2` a. *Add* all components from the repository b. *Add*, *Load*, *Bind*, and *Execute* a script to create the database schemas c. *Load* all components d. *Bind* all local and remote components e. *Execute* all local and remote components
Release 2: Independent Parametric Reconfiguration
The name `dir1` is changed on `N-2` a. *Add*, *Load*, *Bind*, and *Execute* a script on `N-2` that can *Stop* `Dirsync`, rename directory `dir1`, and change `DirsyncSys` parameters b. *Reload parameters* of `DirsyncSys` on `N-2` c. *Execute* component `Dirsync` on `N-2`
Release 3: Dependent Parametric Reconfiguration
`dir2` is added as a new directory to be synchronized in a peer-to-peer fashion a. *Add*, *Load*, *Bind*, and *Execute* a script on `N-1` and `N-2` that can *Stop* `Dirsync` and create properties for directory `dir2` b. *Reload parameters* of `DirsyncSys` on `N-1` and `N-2` c. *Execute* `Dirsync` on `N-1` and `N-2`
Release 4: Independent Implementation Reconfiguration
The file-change algorithm is enhanced to include a checksum of file contents a. *Add*, *Load*, *Bind*, and *Execute* a script on `N-2` that alters the database schema to include a new field `Checksum` b. *Add*, *Load*, and *Bind* a new version of `DirState` on `N-2` having the checksum algorithm c. *Rebind* `Dirsync` on `N-2` to the new version of `DirState`
Release 5: Dependent Implementation Reconfiguration
The format of the command file is changed to include the time of last modification a. *Add*, *Load*, *Bind*, and *Execute* a script that can *Stop* `Dirsync` on `N-1` and `N-2` b. *Unbind*, *Unload*, and *Delete* `Dirsync` on `N-1` and `N-2` c. *Add*, *Load*, *Bind*, and *Execute* a new version of `Dirsync` on `N-1` and `N-2`
Release 6: Independent Interface Reconfiguration
A method is added to `PropertyMgr` for easier access to integer properties a. *Add*, *Load*, and *Bind* a new version of `PropertyMgr` on `N-1` b. *Add* and *Load* a new version of `DirsyncSys` on `N-1` and *Bind* it to new version of `PropertyMgr` c. *Rebind* `Dirsync` on `N-1` to the new version of `DirsyncSys`
Release 7: Dependent Interface Reconfiguration
A subcomponent of `DataChannel` is enhanced for more efficient data transmission a. *Add*, *Load*, and (locally) *Bind* a new version of `DataChannel` on `N-1` and `N-2` b. *Bind* instances of new versions of `DataChannel` to each other on `N-1` and `N-2` c. *Refresh bindings* of `Dirsync` on `N-1` and `N-2`

Table 2. Dirsync Reconfigurations and Associated BARK Commands.

component model, whether it be EJB, .NET, OSGi, or some other "standard" infrastructure. This requires the development of architectural principles that can be instantiated for the particular situation at hand.

Acknowledgments

The work described in this paper was supported in part by the Defense Advanced Research Projects Agency, Air Force Research Laboratory, and Space and Naval Warfare System Center under agreement numbers F30602-01-1-0503, F30602-00-2-0608, and N66001-00-1-8945. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Space and Naval Warfare System Center, or the U.S. Government.

References

1. E.C. Bailey. *Maximum RPM*. Red Hat Software, Inc., February 1997.
2. L.J. Botha and J.M. Bishop. Configuring Distributed Systems in a Java-Based Environment. *IEE Proceedings—Software Engineering*, 148(2), April 2001.
3. R.S. Hall, D.M. Heimbigner, A. van der Hoek, and A.L. Wolf. An Architecture for Post-Development Configuration Management in a Wide-Area Network. In *Proceedings of the 1997 International Conference on Distributed Computing Systems*, pages 269–278. IEEE Computer Society, May 1997.
4. R.S. Hall, D.M. Heimbigner, and A.L. Wolf. Evaluating Software Deployment Languages and Schema. In *Proceedings of the 1998 International Conference on Software Maintenance*, pages 177–185. IEEE Computer Society, November 1998.
5. R.S. Hall, D.M. Heimbigner, and A.L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 174–183. Association for Computer Machinery, May 1999.
6. D.M. Heimbigner and A.L. Wolf. Post-Deployment Configuration Management. In *Proceedings of the Sixth International Workshop on Software Configuration Management*, number 1167 in Lecture Notes in Computer Science, pages 272–276. Springer-Verlag, 1996.
7. InstallShield Corporation. *InstallShield*, 1998.
8. J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.
9. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.
10. R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly and Associates, 2000.
11. K. Ng, J. Kramer, J. Magee, and N. Dulay. The Software Architect's Assistant—A Visual Environment for Distributed Programming. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, 1995.
12. Open Software Associates. *OpenWEB netDeploy*, 1998.