EJB-ARK: Enterprise JavaBean^{TM} Automatic Reconfiguration frameworK

by

M. J. Rutherford

B.S.E., Princeton University, 1996

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2001

This thesis entitled:
`EJB-ARK`: Enterprise JavaBean$^{\mathrm{TM}}$ Automatic Reconfiguration frameworK
written by M. J. Rutherford
has been approved for the Department of Computer Science

Prof. Alexander L. Wolf

Prof. Antonio Carzaniga

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Rutherford, M. J. (M.S., Computer Science)

`EJB-ARK`: Enterprise JavaBean^{TM} Automatic Reconfiguration frameworK

Thesis directed by Prof. Alexander L. Wolf

This project examined the development of the `EJB-ARK`, a management framework for the software deployment life cycle of Enterprise JavaBeans. The over-arching goals of the framework were to allow reconfigurations of EJB systems to proceed with a minimum of disruption, and to minimize the impact on the typical software producer activities. To understand the ways EJB systems evolve, six classifications of system reconfigurations were created. Additionally, enhancements to the existing EJB component life cycle were identified. These enhancements were aimed at making the deployment life cycle of EJBs easier to manage. A prototype `EJB-ARK` system was developed to integrated with JBoss, an open-source EJB server. A sample EJB system, *Dirsync*, was developed to validate the `EJB-ARK` architecture and design. *Dirsync* was designed to have many features that are commonly found in distributed applications, and as a final demonstration the *Dirsync* application was automatically evolved through seven different releases. This study showed that a framework could be developed that would allow a system to be automatically reconfigured with a relatively small impact on the activities of the software producers.

Contents

Chapter

1 Introduction 1

1.1 Software Deployment Life Cycle 1

1.2 Deployment Management of Distributed Systems 3

1.3 EJBs: Distributed Components 4

2 Project Overview 6

3 Related Work 8

4 Enterprise JavaBean^{TM} Background 10

4.1 Overview 10

4.1.1 Stateless Session Beans 11

4.1.2 Stateful Session Beans 12

4.1.3 Entity Beans 12

4.2 EJB Life Cycle 13

4.3 Packaging 15

4.4 The Container 18

4.5 Relationship with JNDI 18

4.6 Why EJBs Were Chosen 18

4.7 EJBs and the Deployment Life Cycle 21

4.7.1 Release 21

4.7.2 Install 21

4.7.3 Activate 21

4.7.4 Reconfigure 22

4.7.5 Update 22

4.7.6 Adapt 23

4.7.7 Deactivate 23

4.7.8 Remove 23

4.7.9 Retire 23

5 Reconfigurations of EJB Systems 24

5.1 Reconfiguration Types 26

5.1.1 Parametric Reconfigurations 26

5.1.2 Implementation Reconfigurations 27

5.1.3 Interface Reconfigurations 27

5.1.4 Single Node vs. Multiple Node 28

5.2 Component Life Cycle Enhancements 28

5.2.1 rebind 29

5.2.2 reloadParameters 29

6 Example Application: Dirsync 30

6.1 Overview 30

6.2 Dirsync Component Architecture 32

6.3 Dirsync Synchronization Protocols 36

6.4 Dirsync Run-time Operation 39

6.4.1 Dirsync.start() 39

6.4.2 Dirsync.synchronize() 41

6.4.3 Dirsync.receive() 43

6.4.4 Dirsync.stop() . . . 46
6.4.5 How Dirsync Receives Data . . . 46
6.5 Summary of "Real-World" Features . . . 49
6.6 Dirsync System Configurations . . . 50
6.6.1 Pre-Installation Assumptions . . . 50
6.6.2 Initial Install . . . 52
6.6.3 Change Node 2 Directory Location . . . 53
6.6.4 Add New Directory to be Synchronized . . . 53
6.6.5 Enhance File Status Algorithm . . . 54
6.6.6 Change Command Format . . . 55
6.6.7 Add Convenience Method for Integer Properties . . . 55
6.6.8 Change Interface of EJBReceiver . . . 56

7 `EJB-ARK` 57
7.1 Overview . . . 58
7.2 Domain Model . . . 61
7.3 Transactions . . . 63
7.4 Application Server Module . . . 66
7.4.1 Composite Deployment Actions . . . 70
7.5 ASM Software Design . . . 71
7.5.1 ASM Package Hierarchy . . . 74
7.5.2 `EJB-ARK` and JNDI . . . 75
7.6 Repository . . . 79
7.7 Workbench . . . 80

8 Deployment Management of Dirsync 84
8.1 Release 1: Initial Installation . . . 84
8.2 Release 2: Single Node Parametric Change . . . 86

8.3 Release 3: Multiple Node Parametric Change 86

8.4 Release 4: Single Node Implementation Change 87

8.5 Release 5: Multiple Node Implementation Change 87

8.6 Release 6: Single Node Interface Change 88

8.7 Release 7: Multiple Node Interface Change 88

8.8 Performance Numbers 88

9 Implications of `EJB-ARK` on Software Producers 90

10 Conclusion 92

Bibliography 93

Tables

Table

4.1 EJB Life Cycle Methods 16

5.1 Reconfiguration Types 26

6.1 Dirsync Roles 31
6.2 Node-Level Roles 36
6.3 Dirsync System Evolution 51

7.1 Remapping of JNDI Names 76
7.2 Binding Setup from ComponentB to ComponentA 77
7.3 Binding Creation from ComponentB to ComponentA 78

8.1 Instruction Timing Results (ms) 89

Figures

Figure

4.1 Simplified EJB Life Cycle (developed from [8]) 14
4.2 EJB Container[11] 19

6.1 Dirsync Component Hierarchy 33
6.2 Inter-node Dependencies 34
6.3 Dirsync Master-slave Synchronization 37
6.4 Dirsync Peer-to-Peer Synchronization 38
6.5 Dirsync.start() Sequence Diagram 40
6.6 Dirsync.synchronize() Sequence Diagram 42
6.7 Dirsync.receive() Sequence Diagram 44
6.8 Dirsync.stop() Sequence Diagram 47
6.9 Receiving Data Sequence Diagram 48
6.10 Initial Dirsync Topology 52
6.11 Final Dirsync Topology 54

7.1 `EJB-ARK` System Overview 59
7.2 Domain Model 61
7.3 `EJB-ARK` Distributed Transaction 65
7.4 Component State Diagram 67
7.5 Running System Component Diagram 72

7.6 ASM Software Packages 73

7.7 EJB-ARK Collection Descriptor 79

7.8 EJB-ARK Script Descriptor 81

7.9 Workbench User Interface 82

8.1 Demo Setup 85

8.2 Release 1 Topology 85

8.3 Release 3 Network Topology 86

Chapter 1

Introduction

This project examined the development of a management framework for the deployment of software systems built using Sun Microsystems' Enterprise JavaBean (EJB) component framework [13]. The "software deployment life cycle" refers to the activities that occur when moving software from a software producer's development environment onto a software consumer's site, the subsequent assembling, configuring and reconfiguring of the software for execution or use in a particular environment, and the end of life activities such as removal and retirement of a software release. EJBs are distributed components, and like all distributed software, they raise the level of complexity of software deployment activities since they must be coordinated between software running on different computers. Due to their inherent complexity, deployment activities of distributed software systems are good candidates for a management and automation framework. The EJB specification [12] provides strict guidelines about how EJB components must be packaged, and how they can reference other components. These guidelines provide most of the structure necessary for an automated system to be able to handle various management tasks.

1.1 Software Deployment Life Cycle

The distinct activities that constitute the software deployment life cycle, as presented by [4], are:

- **release** – As the software producer, packaging all necessary artifacts and configuration descriptions into a form that can be installed at a consumers site.
- **install** – At the software consumer's site, configure and assemble all of the resources necessary to use a released software system. Typically this involves selecting the appropriate pre-defined configuration from the software release that is compatible with the specifics of the consumer's operating environment.
- **activate** – At the software consumer's site, put the installed software into a state so that it can be be executed or used by the software consumer.
- **reconfigure** – At the software consumer's site, modify previously installed software by selecting a different configuration from the existing release.
- **update** – At the software consumer's site, modify installed software by installing a previously unavailable configuration that was newly released by the software producer.
- **adapt** – At the software consumer's site, maintain the consistency of the currently selected configuration.
- **deactivate** – At the software consumer's site, undo any changes to the site that were required to make the software usable or executable.
- **remove** – At the software consumer's site, undo all of the changes that were made to the site due to previous deployment activities. This activity must take special care to handle shared resources properly.
- **retire** – As the software producer, make a specific release unavailable for deployment in the future, and revoke all support for a release.

Some of the most complicated activities in the software deployment life cycle are those that change a system that has initially been installed and activated. These are

reconfigure, **update**, and **adapt**. With these activities there is the added complication of scheduling the changes, and ensuring minimum downtime.

1.2 Deployment Management of Distributed Systems

All software systems that are used in a different environment from that in which they are developed have to participate in the the software deployment life cycle. Deployment life cycle management for stand-alone software is complicated; when dealing with distributed software systems there is the added complication of coordinating deployment activities between multiple systems that are being used by many users simultaneously, and that are potentially geographically dispersed.

Changes to the deployment of a running software system should be done without disrupting the end-user and without losing data. There is the very strict requirement that an error during deployment activities must not render the system unusable when it was working previously. This means that the timing of the configuration is very important. When configuring stand-alone software, the timing is usually left up to the end user, who determines when the system is in a stable state (usually turned off) to do the reconfiguration. Distributed systems are often servicing many end-users at once, so it is not realistic to wait until they are all ready to do a reconfiguration. Again, because distributed systems are often servicing multiple end-users at once, a system must be able to support multiple different configurations simultaneously, providing each end user with the functionality they expect and require.

For critical systems in a business or enterprise, the timely configuration of software systems is often very important. In addition to bug-fixes and functionality enhancements, software reconfiguration can be required to ensure the stability of the system as a whole. This is particularly applicable to high-volume systems or systems operating under strict security policies. In these cases, some external event, perhaps intrusion detection or system load spikes, might trigger a wholesale reconfiguration of a distributed system to provide fault tolerance or enhanced security.

1.3 EJBs: Distributed Components

Rather than tackling the configuration of all distributed systems, this project sought to narrow the field to a reasonable degree and investigate EJBs, Java-based distributed components. Component-based software is common in all types of software development. Common design practices emphasize the decomposition of a complicated software system into manageable pieces that can be divided easily between different teams or developers. Design practices also emphasize the need for building a system out of cohesive, uncoupled building blocks. Component-based software fits these design objectives nicely by allowing the designers to build software modules that map directly to the areas that they delineated during the process of functional decomposition. Typically, designers focus on the interfaces between the components, using sequence diagrams and interaction diagrams to determine which messages need to be passed between components. The contracts that are decided upon also provide a natural basis for unit testing and allow different implementations of components to be substituted as necessary. This separation between component interface and component implementation is one of the primary reasons for choosing to focus on component-based systems for automatic deployment management.

With the system already decomposed into its natural elements, a management framework only has to handle the relationships between different components. One of the major challenges of "black-box" reconfiguration is determining which parts of the software system can reasonably be separated from each other and configured independently. With component-based systems, the responsibility of breaking apart a system has already been tackled during the development process by the designers and architects of the system. Component-based systems provide a natural level of granularity for designers / developers to communicate the independent modules of a software system to an automatic reconfiguration system.

Another convenient characteristic of EJBs is the way in which dependencies between components must be explicitly listed and packaged with the components themselves. This allows a management framework to gain valuable information about the dependencies of the system as a whole.

Chapter 2

Project Overview

The first phase of of this project involved the development of an EJB based, distributed system that could be used for studying how real-world software might need to be reconfigured over time. With such a system in place, a representative set of reconfigurations was established. By classifying the reconfigurations and studying the actions that a human performed to reconfigure the system manually, a set of requirements for a reconfiguration management framework was established.

The second phase of this project dealt with the development of a functioning deployment management framework to validate the ideas that were developed in phase one. This framework was built to handle generic reconfigurations, and its features were demonstrated by automatically applying the reconfigurations identified in phase one while the system was running.

One over-arching goal of this project was to develop an automatic reconfiguration framework that was powerful enough to handle a "real-world" software system, but that would not dramatically increase the level of effort required by the system developers.

Chapter 3 puts the contributions of this project into context by discussing related work in the area of configuration management and distributed configuration management.

Chapter 4 provides an overview of Enterprise JavaBeans, the component framework used in this project.

Chapter 5 presents a model for classifying types of configuration that are needed over the lifetime of a software system.

Chapter 6 outlines the Dirsync application which was developed to illustrate the salient aspects of configuring non-trivial applications.

Chapter 7 discusses the deployment management framework, `EJB-ARK`, that was developed as a proof of concept.

Chapter 8 describes the actual reconfigurations of the Dirsync application by the `EJB-ARK` framework.

Chapter 9 provides some insight into the impact that planning for automatic deployment has on the development of components.

Chapter 3

Related Work

The software deployment life cycle as outlined in Section 1 was presented in [15] and in [4] as part of the University of Colorado Software Dock project. The Software Dock project [4], [5] and [3] is centered around an agent based framework to support negotiations between software producers and consumers on issues surrounding the software deployment life cycle activities. At the heart of the Software Dock is the Deployable Software Description (DSD) which is a standardized deployment schema for describing software systems. The Software Dock has two major components, the Release Dock and the Field Dock that allow agents to attach themselves and help perform the necessary deployment activities. Agents query the Release Dock for information about the software release and at the Field Dock they query the consumer's system for details about the operating environment. This information and the information encoded in the DSD allows agents to manage the deployment activities at both the producer's site and the consumer's site.

A Java-based run-time system known as Jade is presented in [1]. This system is modeled after the Regis run-time environment for the Darwin architectural description language presented in [7]. In these run-time systems, Regis and Jade, components are bound to each other by typed communication objects as described by the Darwin structural description. This is similar in concept to the way EJB package descriptors are kept separate from the executable component code. Jade does not seem to explicitly support many of the software deployment life cycle activities as part of its operation.

Another run-time system based on Java and XML is presented in [2]. This system supports mobile code and allows for fine-grained code updates on running systems. Programs are written in XML and interpreted by a Java-based XML interpreter on the target host. By using XML related technologies such as XPath and the Document Object Model (DOM), a minimal set of new code can be sent to the software consumer. This code is received by the running interpreter, and the running system is subsequently patched. Patches can be applied down to the code statement level, and the interpreter takes care not to update instructions that are actively being interpreted.

The Software Architect's Assistant is presented in [9] and [10]. This is effectively a design-time tool that allows the developer to graphically bind components together to form complete systems. The software that is created is described by the Darwin architectural language and is capable of running in the Regis run-time environment. These articles mention support for managing running systems with the Architect's Assistant, but no details are provided.

Chapter 4

Enterprise JavaBean^{TM} Background

4.1 Overview

Enterprise JavaBeans^{TM} (EJBs) are the cornerstone of Sun Microsystems' Enterprise Java initiative. The Enterprise Java initiative is a suite of technologies designed to provide a standard structure for developing enterprise applications. Mostly, Sun has provided a series of specifications that vendors must comply with to license the Enterprise Java technology. The specifications address issues like database connectivity, transaction support, naming and directory services, messaging services, and distributed components. EJBs are the distributed components. They are deployed[1] within an EJB container which handles a lot of the complexity inherent with multi-threaded, database driven, transactional applications, thereby allowing the developer to concentrate on the business logic.

For a component to be an EJB there must be at least three Java classes that compose it: the Remote interface, the Home interface, and the implementation. The Remote interface defines all of the business methods that are available to clients of a particular component. The Home interface defines various factory methods that can be used by clients to create and retrieve instances of the Remote interface of the bean. Both the Remote and Home interfaces are used by clients of the EJB. The implementation

[1] The term "deploy" is used in the context of EJBs to describe the process by which an EJB server loads an EJB package and passes it to the container to make it available to clients; in the context of the software deployment life cycle, the term refers collectively to all of the activities outlined in Section 1. Hopefully the intended meaning is clear from the context.

class provides implementations for methods defined in both the Home and Remote interfaces. While the Home and Remote interfaces must be available to the EJB client, the implementation class always stays on the server, and calls to its methods are wrapped by the container as depicted in Figure 4.2.

For a client to use an EJB, it must first retrieve an instance of the Home interface from the application server. Usually this is done by querying a naming and directory service with a well known name. Once the client has an instance of the Home interface, it uses any of the factory methods to retrieve instances of the Remote interface. All of the factory methods on the Home interface and the functionality methods available on the Remote interface are RMI remote methods. Method calls on both the Home and Remote interfaces are routed through the EJB container which takes care of security issues, and also makes sure that the server-side implementation is operating in the proper transaction context.

EJBs come in three flavors: stateless session beans, stateful session beans and entity beans. Actually, with the EJB 2.0 specification there are now Message Driven beans which are invoked by the arrival of a message to a specific topic or queue. This project only deals with the first three EJB types since widespread support for Message Driven beans is lacking.

4.1.1 Stateless Session Beans

Stateless session beans are components that do not maintain any state between invocations. Essentially, stateless session beans provide utility functions to clients. Stateless session beans exhibit the best performance in terms of server-side memory usage and speed. Because implementation classes for stateless session beans do not have instance variables, a small number of implementation instances can be used to satisfy all requests. The Home interface for stateless session beans must have a single "create" method that takes no arguments. Stateless session beans are often used as front-ends to complicated business logic since the container handles any transactional issues that arise during their use.

4.1.2 Stateful Session Beans

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. To conserve resources, EJB servers can passivate the stateful session bean implementations while they are associated with a client; this involves the serialization of the data members to secondary storage. Stateful session beans are used to provide more complicated processing to clients. Because each client session is serviced by a different implementation instance, the amount of memory used for active session beans is roughly proportional to the number of concurrent users assuming the clients are using the session bean consistently.

4.1.3 Entity Beans

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 EJB system. Typically entity beans are used as a software representation of a single row of a query into a relational database. Entity beans can have either Bean-Managed Persistence (BMP) or Container-Managed Persistence (CMP). With CMP, the developer provides a mapping of implementation class member variables to database tables and columns, and the EJB container is responsible for loading and storing the data. While this relieves the developer from a lot of tedious query writing, typically only very simple queries involving zero or one joins are suitable for CMP. With BMP, the developer is responsible for handling the specifics of loading and storing data, thereby allowing for the freedom of complicated SQL statements. Each Entity bean instance is uniquely identified by its primary key. Within the container, each primary key maps directly to a single EJB instance, thereby ensuring that concurrent modification of the data is handled properly. Developers have the option of using a standard Java class for a primary key or developing their own composite primary key to handle more complicated data models. Home interfaces for entity beans consist of zero or more "create" methods, zero or more "find" methods, and a "remove" method. All Entity beans must have a "findByPrimaryKey" method which takes an instance of the primary key class as its only argument. As their names indicate, create methods are used to create new entries in persistent storage, find methods are used to look up an instance, or instances that match certain criteria, and the remove method is used to delete the entity from persistent storage.

4.2 EJB Life Cycle

Part of the EJB framework that is specified by Sun Microsystems relates to the various life cycle stages that an EJB implementation instance goes through as it is servicing requests. One major characteristic of EJB server implementations is instance pooling. Essentially, pools of EJB implementation objects are pre-allocated by the server to help manages server resources. In simple terms, when a client makes a request, the container arbitrarily selects an instance from the pool, associates the client's context with it, and calls the appropriate method on the bean implementation class instance.

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. Instance pools for stateless beans are usually quite small. Instances are placed into the pool immediately after they are instantiated and they are removed from the pool only for the duration of a single client method call.

For stateful beans, the life cycle is a bit more complicated since the an instance must be associated either with a particular client across multiple method calls, or with

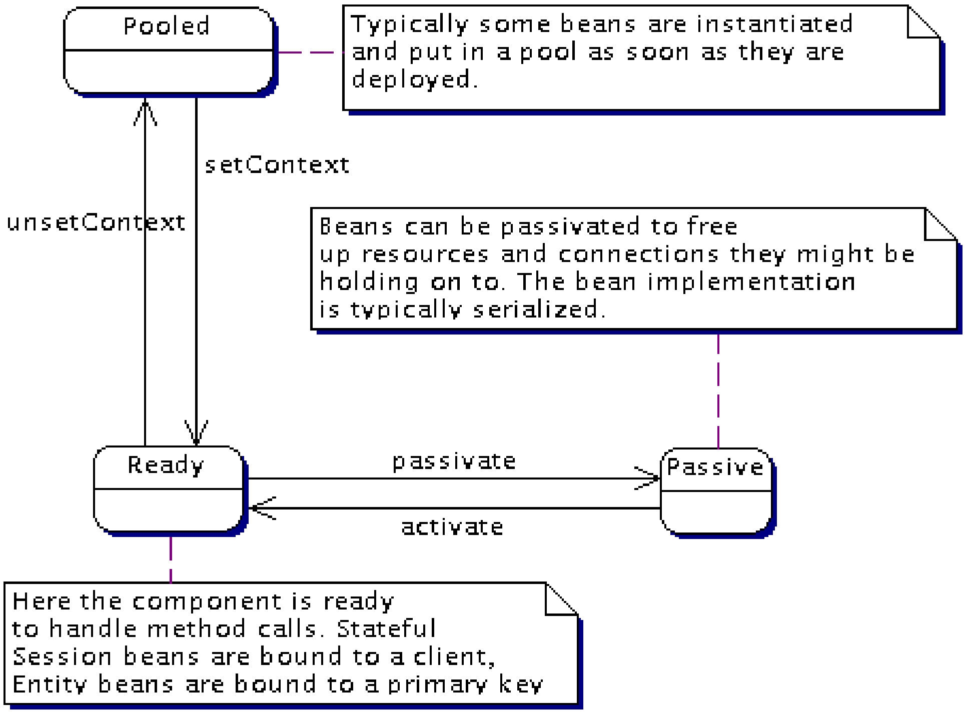

Figure 4.1: Simplified EJB Life Cycle (developed from [8])

a particular persistent entity. The state of the stateful beans is strictly controlled by the container using some required life cycle methods. The diagram depicted in Figure 4.1 shows a simplified life cycle of stateful beans. Before the implementation is bound to a particular or request it is considered to be in the “Pooled” state. When a client calls the create method on the EJB’s home interface (either directly, or indirectly through a finder method on an Entity bean) an instance is picked from the free pool and bound to a particular request (for a Stateful Session bean) or a primary key (for an Entity bean). At this point it is “Ready” to service client requests. For server resource management reasons, an implementation instance may be passivated at any time after it is in the “Ready” state. Best practices dictate that during passivation, an implementation instance releases any limited resources like database connections or file handles. Instances may be serialized to disk or across a network in a clustered environment after they are in the “Passive” state. Once the data for an instance has been serialized to secondary storage, the implementation instance can be returned to the “Pooled” state to service other requests. If a request comes in that needs the data for an instance that has been passivated, the container deserializes the data into any pooled instance, and again returns the bean to the “Ready” state.

EJB servers maintain the state of component implementation instances using methods that are defined by the `javax.ejb.EntityBean` and `javax.ejb.SessionBean` interfaces which all component implementation classes must implement. Table 4.1 describes the four EJB life cycle methods.

4.3 Packaging

Once the classes that comprise an EJB have been developed they 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 Java Archive (JAR) file. Addition-

Method	**Description**
ejbActivate	First method called after a stateful EJB is deserialized from secondary storage. Any resources that are needed by the instance should be retrieved here.
ejbPassivate	Last method called before a stateful EJB is serialized to secondary storage. Any resources that are held by the instance should be released here.
ejbLoad	Valid for entity beans only. This method signals the instance to retrieve current values of its state from persistent storage. This is required for BMP, and can be implemented with CMP to handle special behavior.
ejbStore	Valid for entity beans only. This method signals the instance to save current values of its state to persistent storage. This is required for BMP, and can be implemented with CMP to handle special behavior.

Table 4.1: EJB Life Cycle Methods

ally, some XML descriptors are available at a standard location within the JAR. These component descriptors identify which class files in the JAR are the Home, Remote and implementation classes. The descriptors also typically include the Java Naming and Directory Interface (JNDI) name that the Home interface is to be bound to. The descriptors also include information about what transaction support is needed by the EJB and often access control information is also available. One JAR can contain multiple EJBs, each of which has its own section in the deployment descriptors.

An important part of the standard deployment information that must be provided by developers includes a listing of any other EJBs that a given component depends upon. Each EJB reference that is listed includes the names of the home and remote classes, the JNDI name that the component is deployed to, and the local JNDI alias for the EJB. Containers use this dependency information to make sure that a mechanism is in place to allow an EJB to call other EJBs in the most efficient way possible. If both EJBs involved in a relationship are deployed in the same application server, the container arranges things so that the client EJB is able to make native method invocations. If the EJBs are deployed in different servers, remote method invocations are used. The implementation code for the client EJB uses the local JNDI alias to retrieve instances of home interfaces. The container makes sure that the local JNDI name is linked to the appropriate Home interface implementation.

The packaging specification for EJBs allows multiple EJBs to be packaged together in a single EJB JAR file. An EJB server handles all of the EJBs packaged together as a single application unit. Dependencies between EJBs that are packaged in the same JAR file are known as internal, and the container makes sure the references are wired together properly. The decision of how to package EJBs is up to the software producer, once EJBs are packaged together, they cannot be maintained separately from the other EJBs that they were packaged with. In this project a single EJB JAR file is described as containing a single component, with each EJB in the package representing a different view into the component. The decision of how to package multiple EJBs essentially drives the granularity of how the EJBs can be maintained.

4.4 The Container

The role of the EJB container is to provide coordination and management of EJB instances. Specifically the container must implement processing that is needed by all beans: security management, transaction management, persistence management, and concurrency management. Figure 4.2 shows the path a client request takes through the container and finally to the implementation class for processing.

One feature of the EJB Container that Figure 4.2 does not illustrate is the resource pooling that it provides. In most container implementations, part of the deployment configuration (see Section 4.3) allows developers to dictate how big the instance pools for beans should be. By using pools of pre-allocated objects, containers do not have to incur the overhead of dynamically allocating large numbers of objects on the heap with every client request.

4.5 Relationship with JNDI

As mentioned above, the deployment information for an EJB package typically includes the JNDI name that the Home interface is bound to in the local JNDI service. When an EJB is deployed, the container generates an implementation of the Home interface and binds it to the specified name. Containers also often use different namespaces for each EJB package to support the notion of local aliases for EJB references.

4.6 Why EJBs Were Chosen

Sun's EJB technology was chosen for this project for a number of reasons. The choice of EJBs was in part driven by practical concerns: the specifications and reference implementations are freely available, there is both open-source and commercial support

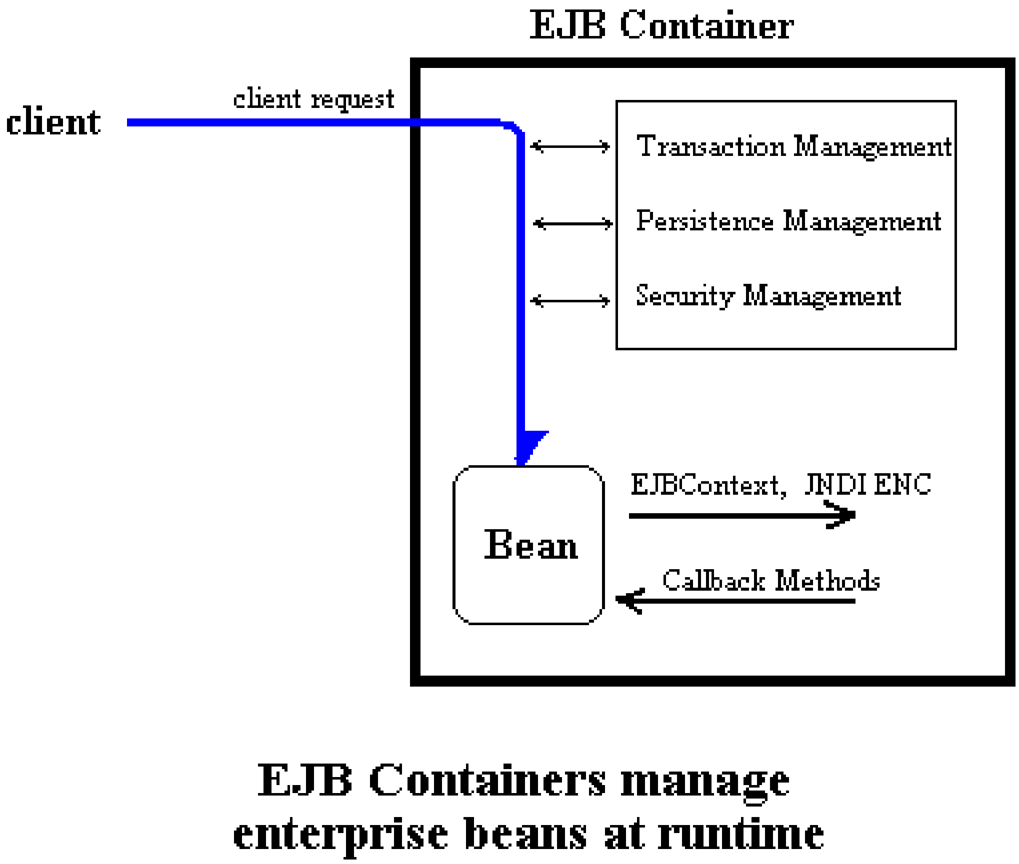

Figure 4.2: EJB Container[11]

for J2EE, a large percentage of Fortune 500 companies use the J2EE framework in their systems, the author is more familiar with EJB technology than any other distributed component frameworks (DCOM, CORBA, etc. . .).

Some of the characteristics of the J2EE framework, and how it is commonly used also made EJBs ideally suited for automatic configuration:

(1) EJBs are distributed components, as such they exhibit many of the management challenges that all distributed software exhibits, but their componentized nature provides enough structure and additional information to make automatic deployment management possible.

(2) The specification clearly forces developers to separate interfaces and implementations, thereby promoting a loose binding between components and their clients.

(3) Part of the descriptive information that EJBs must be packaged with is the listing of other components upon which the EJB depends.

(4) The use of naming and directory services to locate references to instance factories provides a layer of loose binding.

(5) Most J2EE servers are designed to allow dynamic deployment / undeployment of components. This is important for automatic reconfiguration because without it, the servers would have to restarted each time a changes was made, severely inhibiting the ability to reconfigure a system without users perceiving it.

(6) Some of the features of the Java programming language like reflection, serialization, dynamic class loading make the implementation of a deployment management framework possible.

4.7 EJBs and the Deployment Life Cycle

Section 1 detailed the different activities that comprise the software deployment life cycle. This section provides details about the specific types of actions that are taken during the phases of the deployment life cycle for EJBs.

4.7.1 Release

When a software provider releases an EJB-based software system, they typically provide the EJB JAR files that contain both the component software and the descriptions of their contents and relationships. Depending on the decisions made when the EJBs were packaged, there will be one or more EJB JAR packages each containing one or more EJBs. They might also provide database DDL that is compatible with the released components.

4.7.2 Install

With EJBs the install activity involves downloading the EJB JAR files from the release location to the machine that is acting as the EJB server. The software consumer might also have to configure a database with supplied DDL.

4.7.3 Activate

To put an EJB system into a usable state, the software consumer has to make the EJB JAR files available to the EJB server in some manner. This might involve copying the files to a specific directory that is monitored by the EJB server, or it might involve adjusting the configuration of the server so that it recognizes the new component files. Most EJB servers have a mechanism by which new components can be incorporated into the server without having to restart the server, but this is very implementation dependent. Once all of the EJB JAR files have been made available to the server, methods on some of the EJBs might be invoked to perform initialization or customization.

4.7.4 Reconfigure

For the software consumer to select a different configuration from the ones that are available in the software release that they have, they would possibly have to adjust the EJB package descriptors and notify the EJB server to reload them. There could also be some adjustments that have to be made to the database schema or data that go along with this. Depending on the way the EJBs are implemented, components that are dependent on ones that have changed might also have to be reloaded by the server to refresh their bindings.

A reconfiguration could also change the bindings between existing components, in which case the package descriptors of client components would need to be adjusted, and the packages would have to be reloaded by the EJB server.

4.7.5 Update

To update the EJB system, the software consumer would have to download any new EJB JAR files, make them available to the EJB server, and possibly force the server to reload some existing component packages to make sure the bindings were current. If the newly released component packages were replacements for existing components, then the previous version of the component would have to be removed before the new version could be loaded by the server.

A update could also require changes to the other resources that are used by the installed components, including changes to a database schema or content, or other data files.

4.7.6 Adapt

If an EJB system becomes out of sync with its current configuration, any number of steps might need to be taken to correct it. These might include adjusting the database schema or data, triggering the EJB server to reload component packages, adjusting the bindings between components, or any number of application specific changes.

4.7.7 Deactivate

To deactivate an EJB system, bindings between components would have to be removed, or component would be removed from the system. Database data might have to be deleted, or the entire database might have to be dropped.

4.7.8 Remove

To remove a release from the consumer's system, all the EJB JAR files related to the system would have to be removed, and any other artifacts like database DDL or data files would also have to be deleted.

4.7.9 Retire

For the software producer to retire a software release, all the artifacts that comprised the software release would have to be removed from the locations that software consumers retrieved them from.

Chapter 5

Reconfigurations of EJB Systems

Section 4.7 listed the specific steps that often need to be taken during the phases of the software deployment life cycle for EJBs. Many of the actions that were required on the part of the software consumer were common across multiple phases:

- Retrieve a new component package from the software consumer (**install** and **update**).
- Load a component package into the EJB server (**activate**, **reconfigure**, **update** and **adapt**).
- Unload an undesired version of a component package from the EJB server (**update**, **deactivate** and **reconfigure**).
- Reload a component package into the EJB server to freshen its bindings to other components (**reconfigure**, **update** and **adapt**).
- Create or update a database's schema or data, or other content files (**activate**, **update**, **reconfigure** and **adapt**).

One of the major problems with these actions is that in many cases they are very heavy-handed. This is especially true of the actions that are taken during deployment phases that occur after the software has been activated and is being used. Having to reload an entire component package just to make sure the bindings are updated makes the entire deployment process for EJBs very invasive.

Having to shut down some or all of an EJB-based system is highly undesirable. In most cases, EJB technology is chosen because of its robust nature, its tight integration with a transaction server and security manager, and its scalable design. These things are especially important in high-volume, high-availability applications where managing downtime is extremely important.

To a certain extent, this heavy-handedness is dictated by the way the component package descriptors are used to specify the dependencies between EJBs. Included in the package descriptor for a component is the well-known naming service name of all other referenced components (that are not contained in the same package as the first component). When the referenced component is loaded by its EJB 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. This means the system will effectively be unable to satisfy requests for a short time. If 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 and they must be reloaded by their EJB servers; again resulting in downtime.

To better understand how reconfiguring or updating a system could be made less invasive, six different types of deployment changes were examined. Additionally, some enhancements to the basic EJB life cycle that would reduce the invasiveness of reconfigurations are also examined.

	Single Node	Multiple Nodes
Parametric	Any pre-planned configuration that requires changes only on a single node.	Any pre-planned configuration that requires changes on multiple nodes transactionally.
Implementation	Any change to the implementation of a component that does not require an implementation change to its clients.	Any change to the implementation of a component that does not require an implementation change to its clients, but that must be coordinated with a similar change on another node transactionally.
Interface	Any change to a component that requires a corresponding change to its clients' (who are all local) implementations.	Any change to a component that requires a corresponding change to its its clients' implementations where some of the clients are on a different server.

Table 5.1: Reconfiguration Types

5.1 Reconfiguration Types

The various actions outlined in Section 4.7 were categorized into three major groups: parametric reconfigurations, implementation reconfigurations, and interface implementations. Since EJBs are a framework for distributed computing, each of these reconfigurations could potentially require coordination across multiple nodes. Table 5.1 summarizes the features of each type of reconfiguration for single nodes and for multiple nodes.

5.1.1 Parametric Reconfigurations

A parametric reconfiguration is essentially a catch-all for any reconfigurations that the software system was designed to be able to handle. One of the most common ways of planning for change in software systems is to have certain options specified as parameters through some external means like a property file or database entries. By doing this, changes can be made to the software system without having to change any of the executable software components that comprise the running system.

At first glance it might seem that the types of reconfigurations that can be accomplished with parametric changes would only be quite simple. This is not the case. For instance, a single boolean parameter could be used to signal to the system to use an entirely different set of components to perform some task.

This type of reconfiguration might have to be coordinated across multiple nodes when the change must be made on all nodes at the same time. An example of this might be changing a parameter that causes all communications to use a preconfigured encryption scheme. If the change is made on some nodes before others communications would be disrupted.

5.1.2 Implementation Reconfigurations

In Section 4, the separation of interface and implementation inherent in EJBs was outlined. In simple terms, and implementation reconfiguration is one in which a change is made to a component that does not require its clients to change their implementation. This includes changes in method signatures and also changes in the interpretation of passed data.

It should be pointed out that a simple change in implementation may effect the **contract** between two components. For instance, a component may expose a method that takes a single string as its argument. The format of this string is important, and an implementation change in the component may change the expected format of this argument. This would require all client components to also change their implementation to conform to the new contract, even though the exposed interface never changed.

It is possible that the implementation change is one that must made to multiple instances in a coordinated fashion, because the nature of the implementation change is such that data passed between the nodes is formatted / interpreted differently.

5.1.3 Interface Reconfigurations

An interface reconfiguration is any configuration change that requires clients of the changed component to also be reconfigured. This change might be in the method signatures that are exposed by the component, or it might just be in the contract of any data that is passed between the component and its client components.

5.1.4 Single Node vs. Multiple Node

The three reconfiguration types mentioned above can all occur within the components on a single node or across multiple nodes. The classification of a particular reconfiguration as single-node or multiple-node depends on the whether the reconfigurations need to be synchronized or not. In the cases of parametric reconfigurations and implementation reconfigurations, this depends heavily on the nature of the changes that are being made. For instance, any parametric or implementation change that results in a change of the contract between components that are running on different nodes requires coordination of the changes on both of the nodes. In fact, the changes must 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 are many parametric and implementation changes that can be made 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.

5.2 Component Life Cycle Enhancements

As outlined in Section 4, containers shepherd the components they are managing through the various phases of the EJB life cycle by invoking required life cycle maintenance methods on the implementation class of the component (see Table 4.1).

In an effort to make EJB systems less prone to downtime during reconfiguration, two additional life cycle methods with specific semantics are proposed. The adoption of these methods by component developers would give management tools much tighter control over the operation of the components after configuration changes are made.

Two common characteristics of EJB components are that they often are clients of other EJB components, and they often load parameters from some external source like a database or property file. When changes are made to their parameters or to the components that they depend on, they need to be notified. With conventional EJB servers the only way to force an EJB to freshen its bindings is to force the server to reload the entire component package; same for the loading of parameters. The other alternative is to have each EJB instance re-establish its bindings and parameters before each use, however this could severely degrade performance since opening a file or database connection is slow, as is doing lookups in a remote naming service.

The two proposed life cycle methods are `rebind` and `reloadParameters`

5.2.1 rebind

The purpose of this method would be to notify the component when it needed to look up new component factories (Home interfaces) from the naming service for all of the other components that it depends on. In the absence of this method, the component implementation instances could happily use the statically cached reference to the component factories.

The utility of this method depends on the specifics of the component implementation. If Home interface references are not cached for performance, or a component doesn't depend on any other components, then this method would not need to be implemented. It should be noted that care needs to be taken to synchronized access to statically cached Home interfaces.

5.2.2 reloadParameters

The purpose of this method would be to notify a component when it needed to re-read its parameters. This would allow the component to statically cache parameter values for efficiency, but still be notified when they needed to reload them.

Again, the utility of this method depends on how a component is implemented.

Chapter 6

Example Application: Dirsync

The Dirsync application was developed to help ensure that the design of `EJB-ARK` was powerful enough to handle non-trivial systems. It is a component-based distributed system build out of a number of interrelated components that together provide a service for synchronizing directories on remote machines.

Multiple versions of Dirsync were developed to provide a representative sampling of how such a distributed component-based system might evolve over time. These reconfigurations of Dirsync were used to make some generalizations about different types of reconfigurations that are likely to exist across most applications, they are detailed in Section 6.6. The reconfigurations of Dirsync were also used to validate the requirements for the `EJB-ARK` system described in Section 7.

This section describes the functionality and design of Dirsync.

6.1 Overview

Dirsync is a component-based application that synchronizes the contents of shared directories on remote computers. Each instance in a Dirsync system is configured with its shared directories, and the hosts that the directories are automatically synchronized with. Dirsync maintains relationships between remote instances on a per-directory basis, allowing for both master-slave and peer-to-peer relationships for each directory that is shared. With Peer-to-peer relationships, one computer is designated as the "peer-

Role	Description
master	The master pushes changes to its slaves. The master initiates all communications. Any changes made in the shared directory on the slave will not be synchronized.
slave	The slave receives changes from its master. The slave passively accepts all changes, even if it means that a local change will be lost.
peer-master	The peer-master initiates all communications with its peer-slave. Changes that occurred locally are sent to each peer-slave. Changes that occurred on the peer-slave are sent back for handling by the peer-master.
peer-slave	The peer-slave receives changes from its peer-master. Modifications on the peer-slave are sent to the peer-master once it has initiated a synchronization.

Table 6.1: Dirsync Roles

master" and this instance drives the timing of the synchronization and determines how any file conflicts should be resolved. The Dirsync system runs as a service on all the instances that are involved in synchronization. The synchronization of each directory is driven by a timer service running for each directory that the local node is either a **master** or a **peer-master** for.

For each directory that is shared, a particular machine may play one of four roles in the synchronization. Table 6.1 describes these.

The Dirsync system is composed of a set of coupled components that expose well-known interfaces to each other through the standard EJB home and remote interfaces (see Section 4). The highest-level component, appropriately called *Dirsync*, coordinates the activities of the other components to ensure proper synchronization of the directories. Most of the components that comprise the Dirsync 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 the functionality of Dirsync is fairly straightforward, the complexity that exists arises out of the relationships between the different components (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 fairly representative configuration challenge. Section 6.5 outlines the specific characteristics of Dirsync that make it similar to many commercial applications.

6.2 Dirsync Component Architecture

As mentioned above, Dirsync is comprised of a number of coupled components. Figure 6.1 shows their dependencies. An arrow pointing from *Component A* to *Component B* shows that *Component A* **uses** *Component B*. Moving from bottom to top in the dependence graph, the functionality that the components provide becomes more specific to the Dirsync application. Most of the components shown in Figure 6.1 is actually comprised of multiple EJBs. Multiple tightly coupled EJBs are often packaged as a single component.

The following list gives some idea of the responsibilities of each of the components. The components are listed from least- to most-relevant to the Dirsync functionality

- *Sequence* – This utility component is an interface to a generic database sequence. It is used to generate new numeric primary keys for entity beans. It is backed by a database `SEQUENCE`, and the name of the particular sequence to use must be provided by the calling component. *Sequence* is only used by *Logger*, since the other entity beans used in the Dirsync application have non-numeric primary keys.

- *Logger* – This component is used by all the other components to record status, debug, and error messages. Messages are persistently stored in the local database. The external interface to this component is through the *Logger* EJB.

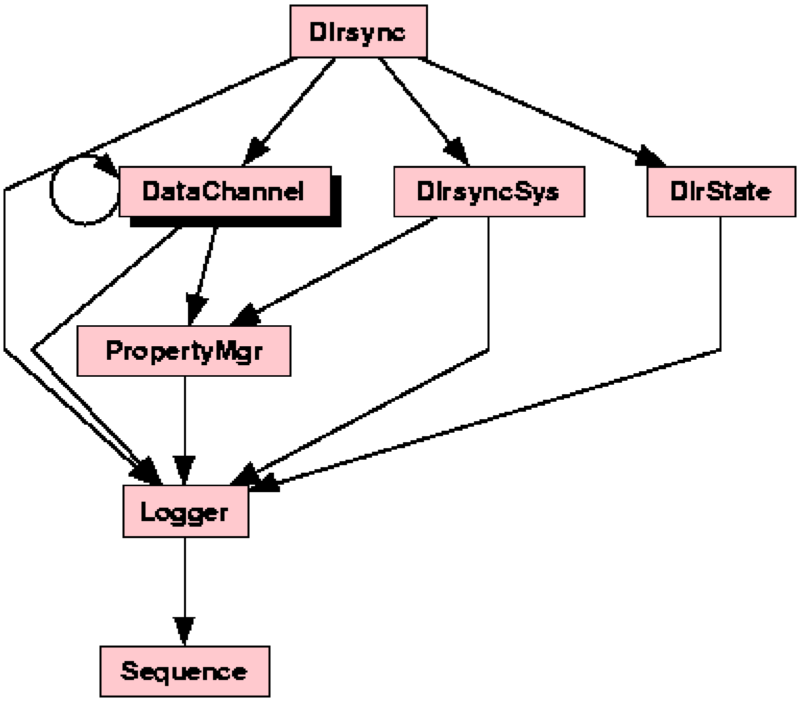

Figure 6.1: Dirsync Component Hierarchy

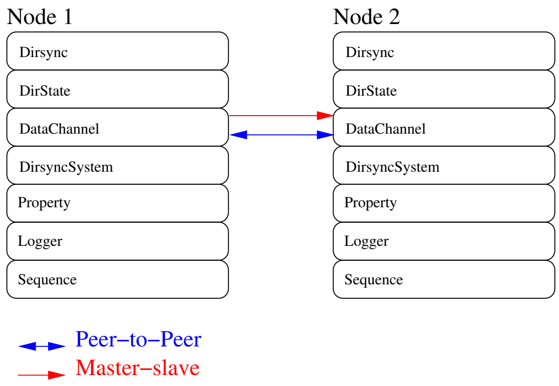

Figure 6.2: Inter-node Dependencies

This stateless session bean uses the *LogEntry* entity bean to handle persistence of messages behind the scenes. The *LogEntry* component uses *Sequence* to generate its numeric primary keys.

- *PropertyMgr* – The *PropertyMgr* component provides its clients with a means for retrieving and altering their configuration parameters. The parameters are stored persistently in a local database using the internal *Property* entity bean. The *PropertyMgr* bean provides a more convenient interface to the persistent properties. All properties are stored as name-value pairs.

- *DirState* – This component manages the persistence of the state of a directory on the local file system. A directory consists of the files and other directories that it contains. Modification times and file sizes are stored for all files in a directory. In later versions of the Dirsync application, MD5 check-sums are also saved. This component is able to (recursively) determine which files have changed since the last time it was updated. The external interface to this component is the *DirState* session bean; internally this uses the *FileEntry* entity bean to manage persistence of the state of a single file. *DirState* provides interfaces to update an entire directory structure and a single file.

- *DataChannel* – This is a component that facilitates sending and retrieving data from a remote machine. This component is designed to have sub-components that are used to provide the specific mechanisms for sending/receiving data: EJB/RMI, connection-based sockets, connection-less sockets, HTTP tunneling, etc... Currently, only the EJB/RMI transport mechanism is implemented. A DataChannel component talks directly to a remote DataChannel component, using the specified channel implementation. Figure 6.2 shows the dependencies between the DataChannel components on different nodes in both a master-slave and a peer-to-peer situation. The data is moved from a local file on the sending machine, to a local file on the remote machine. The data package that is received is stored in a temporary location on the file system and an entry for it is added to a queue. The higher level *Dirsync* component constantly queries the *DataChannel* receiving queue to see if any packages have arrived for it. Section 6.4.5 has a more complete discussion of how data is sent and received by the *DataChannel* component.

- *DirsyncSys* – This component represents a model of the current configuration of the entire Dirsync system, providing functionality for reasoning about hosts, shared directories, relationships, etc... This model is constructed by querying the *PropertyMgr* component for all the Dirsync related. properties. This component validates the system configuration, and provides convenience functions for retrieving information about the system. This component is used extensively by the *Dirsync* component.

- *Dirsync* – This component is the highest-level component for the Dirsync system. It manages the sequencing and data flow between the other components. This component first connects to a *DirsyncSys* component. Once the system configuration has been validated, timers are started for any relationships in

Roles that A plays with B	Aggregate Relationship
master only	Node A is a master of Node B
master and slave	Nodes A and B are treated as peers.
peer only	Nodes A and B are treated as peers.

Table 6.2: Node-Level Roles

which the local host is master. The call-back method that is provide to the timers that are started is a method on the *Dirsync* session bean that starts the process of synchronizing a specific directory. There is also a receive method that is triggered by a timer to check the *DataChannel* queue for any newly arrived package. Section 6.4 has a more complete discussion of the operation of the *Dirsync* component.

The topology of a network of computers that are synchronizing one or more directories using Dirsync is maintained persistently in the form of properties on each instance in the system. The *DirsyncSys* component provides a way to reason about the system configuration. Both parties in a Dirsync relationship must agree on the roles that each is playing for the directory in question.

Figure 6.2 shows the dependencies between two DataChannel components on different nodes for master-slave and a peer-to-peer situation. In this case, the relationship is determined at the node level, not at the directory level.

6.3 Dirsync Synchronization Protocols

When *Dirsync* synchronizes the contents of a shared directory between two remote nodes, a series of messages are passed between the two Dirsync instances, each using the *DataChannel* component. When synchronizing a directory with a master-slave relationship, there is only uni-directional data flow, from the master to the slave. As shown in Figure 6.3, the messages sent from master to slave basically tell the slave node what changes to make to its local file system. The master uses the *DirState* compo-

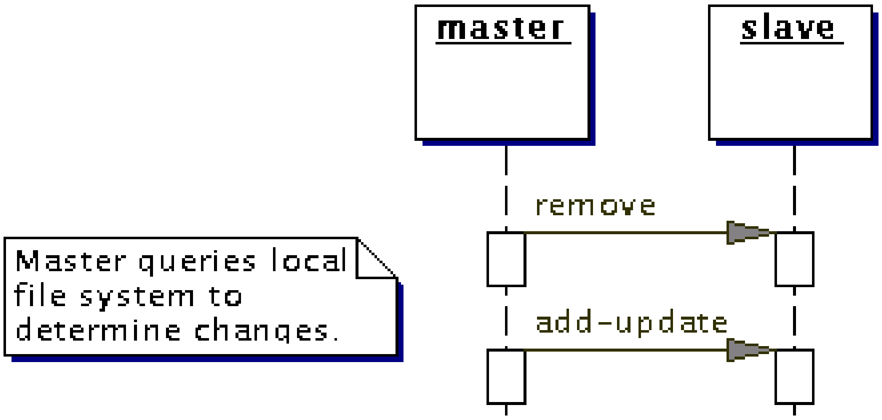

Figure 6.3: Dirsync Master-slave Synchronization

nent to determine which files have changed locally, and sends the necessary commands sequentially to the slave.

The interaction between the peer-master and peer-slave during a directory synchronization is much more complicated, as depicted in Figure 6.4. When a synchronization is triggered by a timer, the peer-master first sends the `SEND_CHANGES` message to the peer-slave. Upon receiving this, the peer-slave creates a list of local changes that have occurred since the last synchronization and sends them to the peer-master with the `CHANGES` message. When the peer-master receives the `CHANGES` message from the peer-slave, it compares the peer-slave changes with those that have occurred locally. If any conflicts are detected, the master uses the configured conflict resolution mechanism (keep bigger, keep smaller, keep older, keep newer, keep master, keep slave) to determine which version of the file should be kept. The peer-master removes any unnecessary changes from the list sent by the peer-slave, and sends the pruned list back to the peer-slave with the `MAKE_CHANGES` message. Upon receiving this, the peer-slave makes all the necessary changes. The peer-master also sends its changes to the peer-slave.

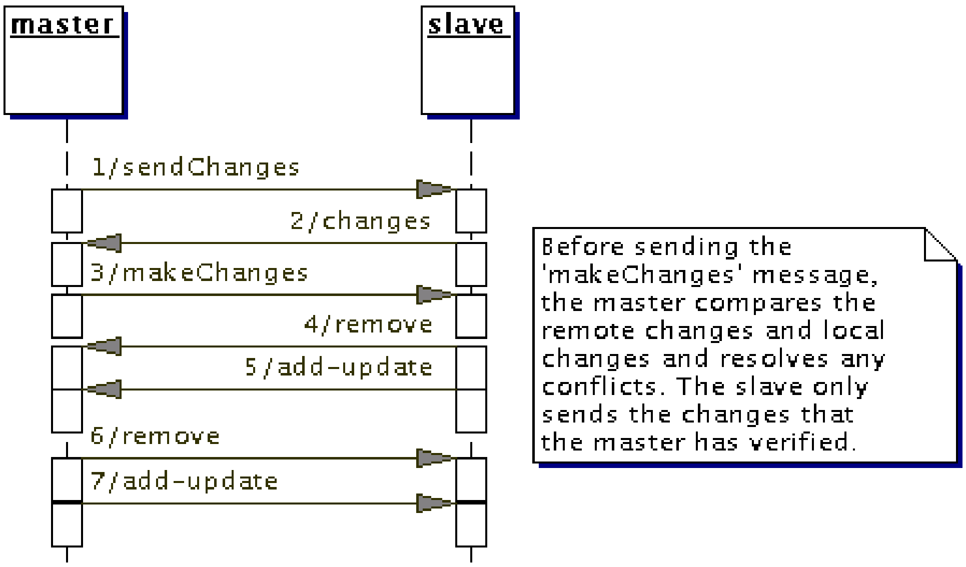

Figure 6.4: Dirsync Peer-to-Peer Synchronization

6.4 Dirsync Run-time Operation

There are essentially four entry points into the Dirsync application: start, stop, synchronize, and receive. The start and stop methods are typically called by an exterior client, while the synchronize and receive methods are usually executed as a call-back for a timer. The following sections provide a detailed look at how each of these methods work.

6.4.1 Dirsync.start()

Figure 6.5 shows the sequence diagram for the

implementation of the start method. In addition to the *Dirsync* component, the *Timer* object and the *DirsyncSys* component are also involved. Note that the *Timer* object does not show up in Figure 6.2. The *Timer* object is actually an internal session bean that is bundled with the *Dirsync* component.

In message 1.1 Dirsync calls `Timer.startReceiveTimer`, which starts a special timer thread that will repeatedly invoke the `receive` method on the *Dirsync* component (see Section 6.4.3). Message 1.2 queries DirsyncSys for all the directories over which the local node is a master. This list of directories will therefore contain all directories for which the local node is either **master** or **peer-master**. Message 1.3 starts a synchronization timer for each master directory. The directory alias is passed to the timer as an argument for the `synchronize` call-back (see Section 6.4.2). Note that while a node may have many different relationships for a given directory, only one timer is started. This is because the synchronization interval is defined on a per-directory basis, not a per-relationship basis.

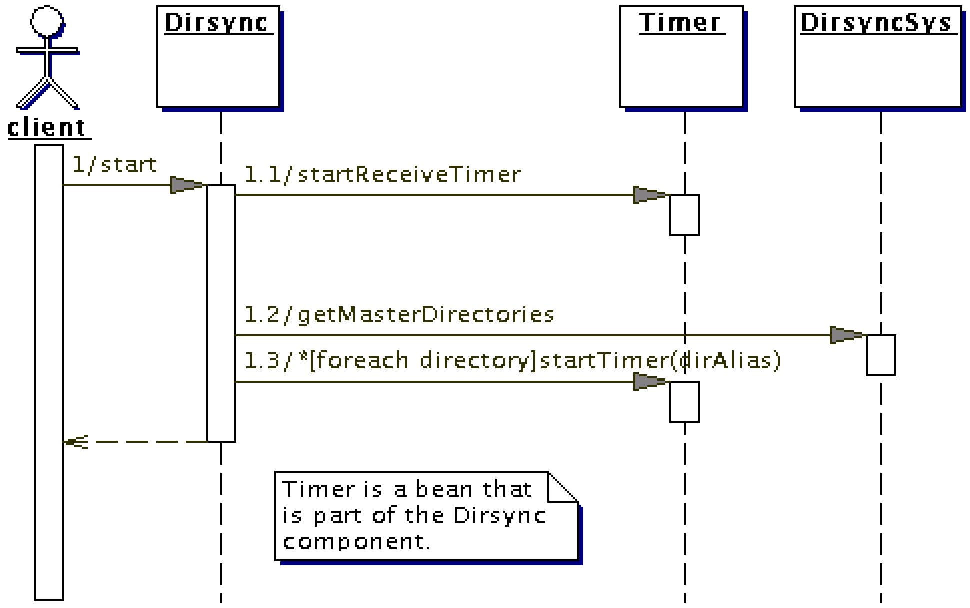

Figure 6.5: Dirsync.start() Sequence Diagram

6.4.2 Dirsync.synchronize()

Figure 6.6 shows the sequence diagram for the implementation of Dirsync's synchronize method. This method is typically invoked as the call-back to a timer initiated in the start method (see Section 6.4.1). The *Dirsync* component uses *DirsyncSys* to reason about the configuration of the node, *DirState* to determine which files have changed, and *DataChannel* to send commands and file data to the remote node.

The timer passes the directory alias to the synchronize method; this is the same information that was provider to the timer when it was initiated. The first thing that Dirsync does is retrieve all the relationship information for the directory in question. Dirsync is only really interested in relationships for which the local node is either master or peer-master. Messages 1.2 and 1.3 show the different paths that are taken for a master-slave relationship and a peer-to-peer relationship.

For master-slave relationships, the master first queries the local file system path to the directory in message 1.2.1. With this absolute path, *Dirsync* uses the *DirState* component to determine what changes have been made locally. For each local change, Dirsync sends a command to the *DataChannel* component which will dispatch it to the slave node. The commands that are sent to the slave node are effectively the list of steps it needs to take to get in sync.

For peer-to-peer relationships, the initial synchronize handling is much simpler. To kick off the peer-to-peer protocol, *Dirsync* only has to dispatch the `SEND_CHANGES` message to the peer-slave using the *DataChannel* component. The bulk of the work for the peer-to-peer synchronization is done during Dirsync's receive method (see Section 6.4.3) in response to the messages sent back and forth.

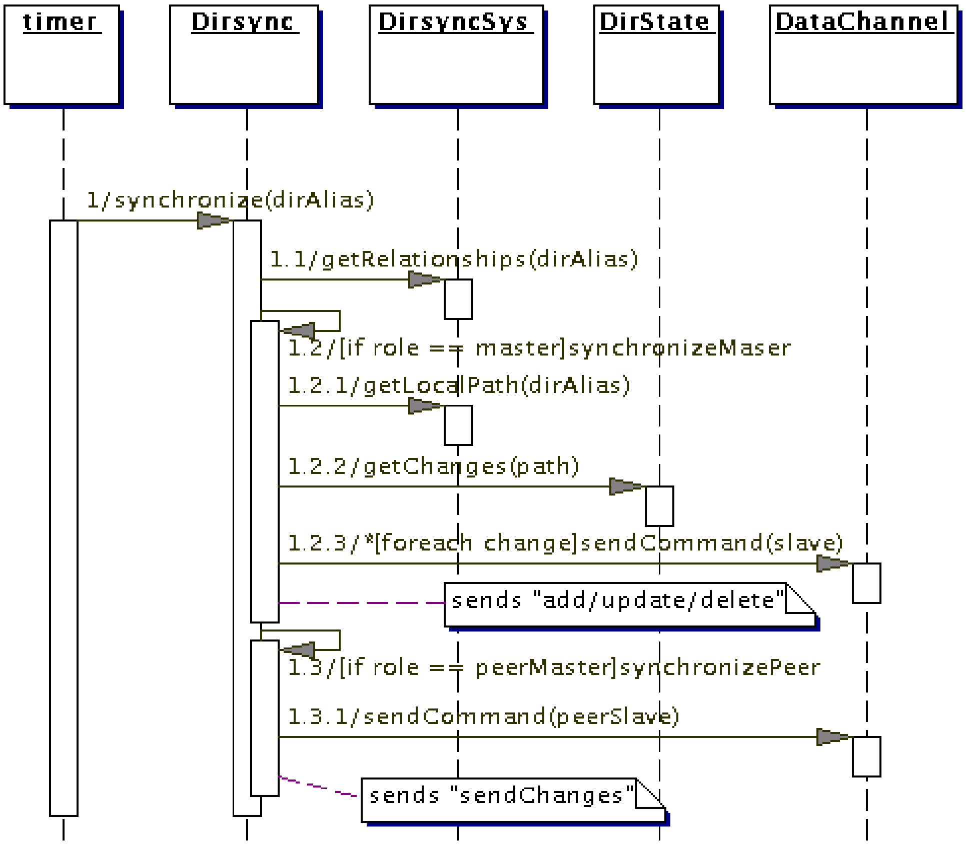

Figure 6.6: Dirsync.synchronize() Sequence Diagram

6.4.3 Dirsync.receive()

Figure 6.7 shows the sequence diagram for the implementation of the receive method. The receive method is invoked continuously by a special thread. When data is received (see Section 6.4.5) the *DataChannel* component stores each bundle in a queue, keyed by the bundles destination. For Dirsync, all bundles have the "Dirsync" destination.

In Message 1.1 *Dirsync* polls *DataChannel* for any newly arrived bundles. *Dirsync* handles the bundles differently depending on their type. Figure 6.7 only depicts the handling of the `ADD_FILE` bundle, but this is fairly representative of all of the different messages detailed below. The `ADD_FILE` bundle contains the following data:

- Source hostname – The host name of the instance that is sending the file.
- Destination hostname – The host name of the instance that is receiving the file. This is included to facilitate a single Dirsync instance that is known by multiple different names.
- Directory alias – This is the common name shared between Dirsync instances for the directory being synchronized.
- Relative file path – This is the file path, relative to the root of the directory being synchronized.
- Last modification date – In later versions of Dirsync (see Section 6.6), the modification time of the file is included to allow for a more exact synchronization of the directories.
- File data – The contents of the file.

In Message 1.2.1 *Dirsync* uses the source and destination hostnames, the directory alias, and the relative file path to determine the absolute path of the file being received

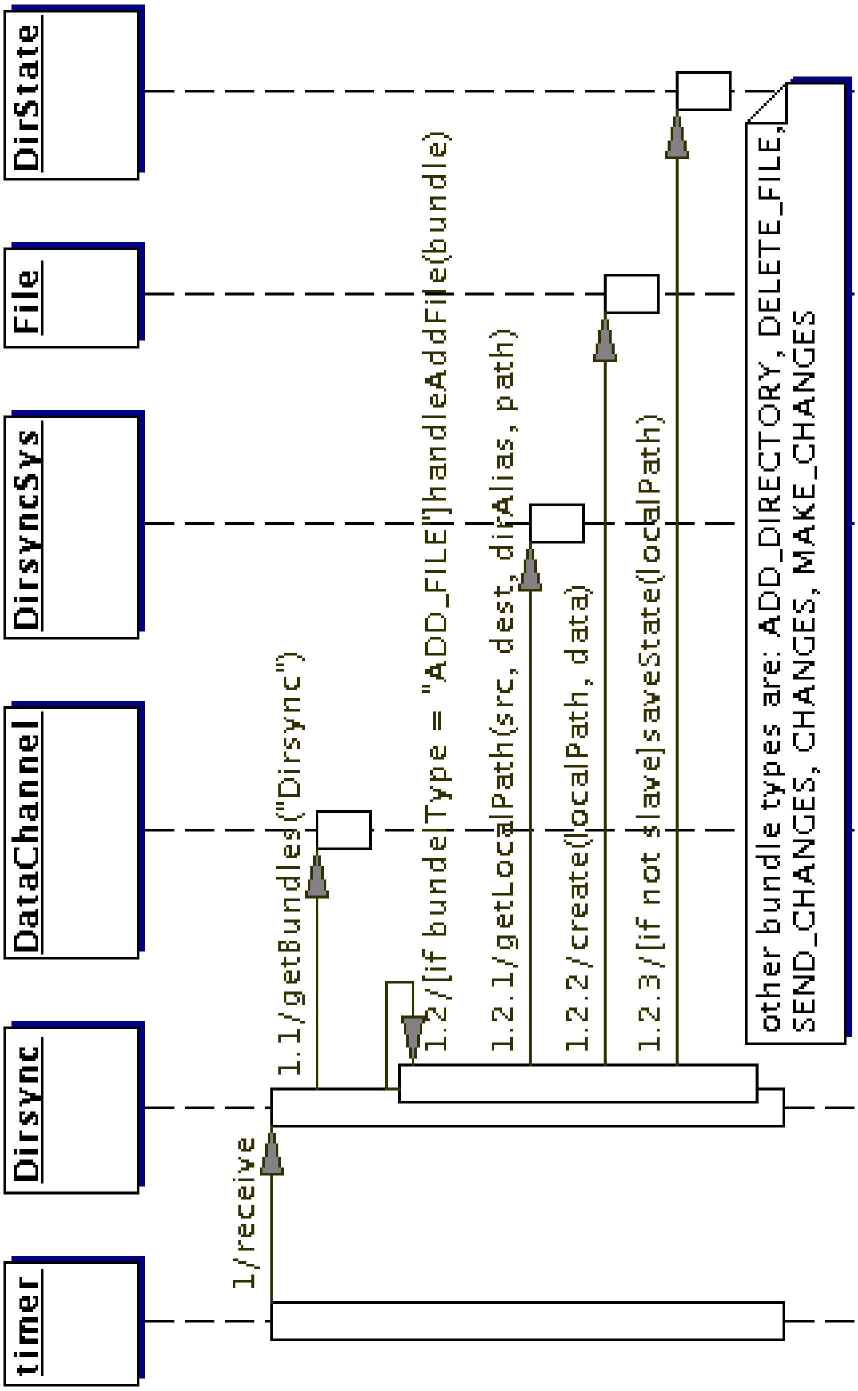

Figure 6.7: Dirsync.receive() Sequence Diagram

on the local file system. It then creates or overwrites the local file with the data that is included in the bundle as shown in Message 1.2.2. If the receiving instance is not a **slave** the state for the new file has to be updated so that the file is not automatically synchronized during the next synchronization. To do this *Dirsync* sends Message 1.2.3 to *DirState* to notify it to store the new information for the file.

In addition to `ADD_FILE`, there are five other messages listed below that *Dirsync* recognizes. All of these bundles contain the source and destination hostnames and the directory alias. Some of the messages also contain additional information that is needed for the bundle, as detailed by the following list:

- `ADD_DIRECTORY` – This bundle also contains the relative path of the directory to be created. Upon receiving this, the instance will create the specified directory.

- `DELETE_FILE` – This bundle contains the relative path of the file or directory to be created. When *Dirsync* receives this, it removes the file or directory from its local file system.

- `SEND_CHANGES` – This bundle should only be received by a **peer-slave**. This bundle contains no additional data. It signals the receiving **peer-slave** to send the list of changes back to the source host.

- `CHANGES` – This bundle is only received by a **peer-master**. In addition to the standard data, this bundle also contains a list of changes to the file system on the remove **peer-slave**. Upon receiving this, *Dirsync* compares the remove changes with changes seen on the local file system, and determines the final list of changes the **peer-slave** should make.

- `MAKE_CHANGES` – This bundle is the final bundle in a peer-to-peer exchange (see Figure 6.4). In addition to the standard information, this bundle contains the pruned list of changes that a **peer-slave** should make.

6.4.4 Dirsync.stop()

Figure 6.5 shows the sequence diagram for the implementation of the stop method. In addition to the *Dirsync* component, the *Timer* object is also involved.

The implementation of the `stop` method is fairly straightforward. Message 1.1 stops each of the synchronization timers that were started during the `start` method (see Section 6.4.1). These timers are not stopped immediately, since they might be in the middle of a synchronization call, however, they will not initiate any new synchronizations after the `Timer.stopTimer` method has been called. Message 1.2 stops the receive timer, ensuring that no new changes will be processed. It is interesting to note that the *DataChannel* component is not explicitly stopped when the Dirsync application is stopped, so incoming messages will be received by the *DataChannel*, but will not actually be processed until *Dirsync* is started again.

6.4.5 How Dirsync Receives Data

In the discussion above, the sending and receiving of data is abstracted at the *DataChannel* level. Figure 6.9 provides a more detailed view of how *Dirsync* sends and receives data. When another component uses *DataChannel* to send data, the *DataChannel* component first determines which implementation to use by querying the value of the `datachannel.transport` property from *PropertyMgr*. Currently, there is only an EJB implementation available, so this is used. *DataChannel* hands off the data to be sent to its internal *EJBSender* component. In message 1 of Figure 6.9 *EJBSender* connects to the *EJBReceiver* component on the remove host, and sends it the data bundle. Every data bundle has a destination, which essentially identifies which application should handle then bundle, and a bundle-type, which can be used by the handling application to differentiate between its own bundle formats. For all of the bundles sent by the *Dirsync* application, the destination is "Dirsync" and the bundle-types are one

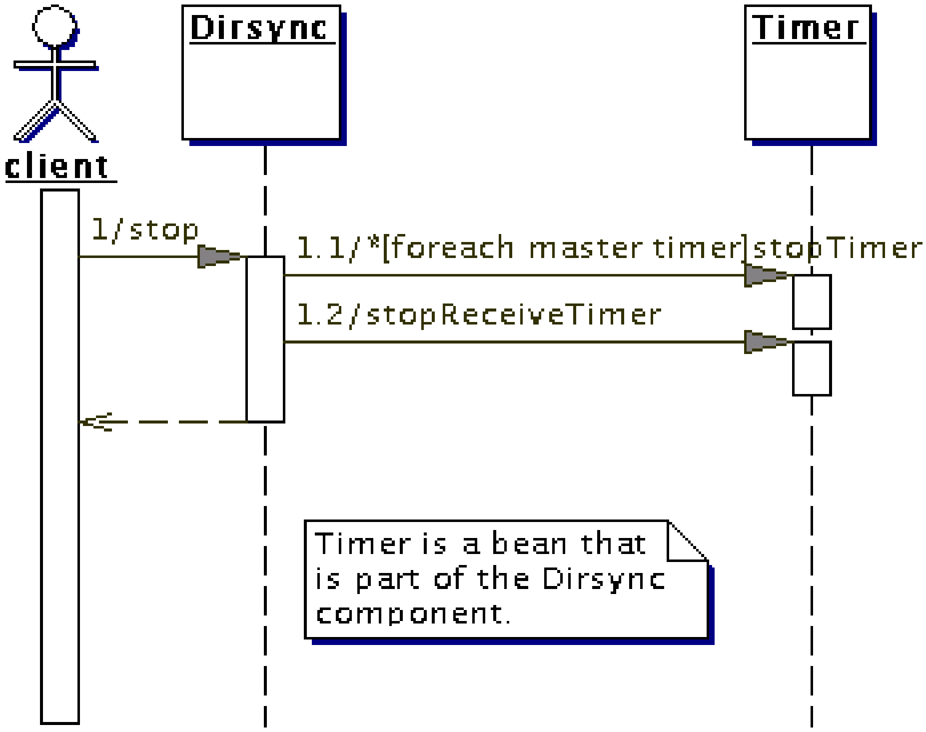

Figure 6.8: Dirsync.stop() Sequence Diagram

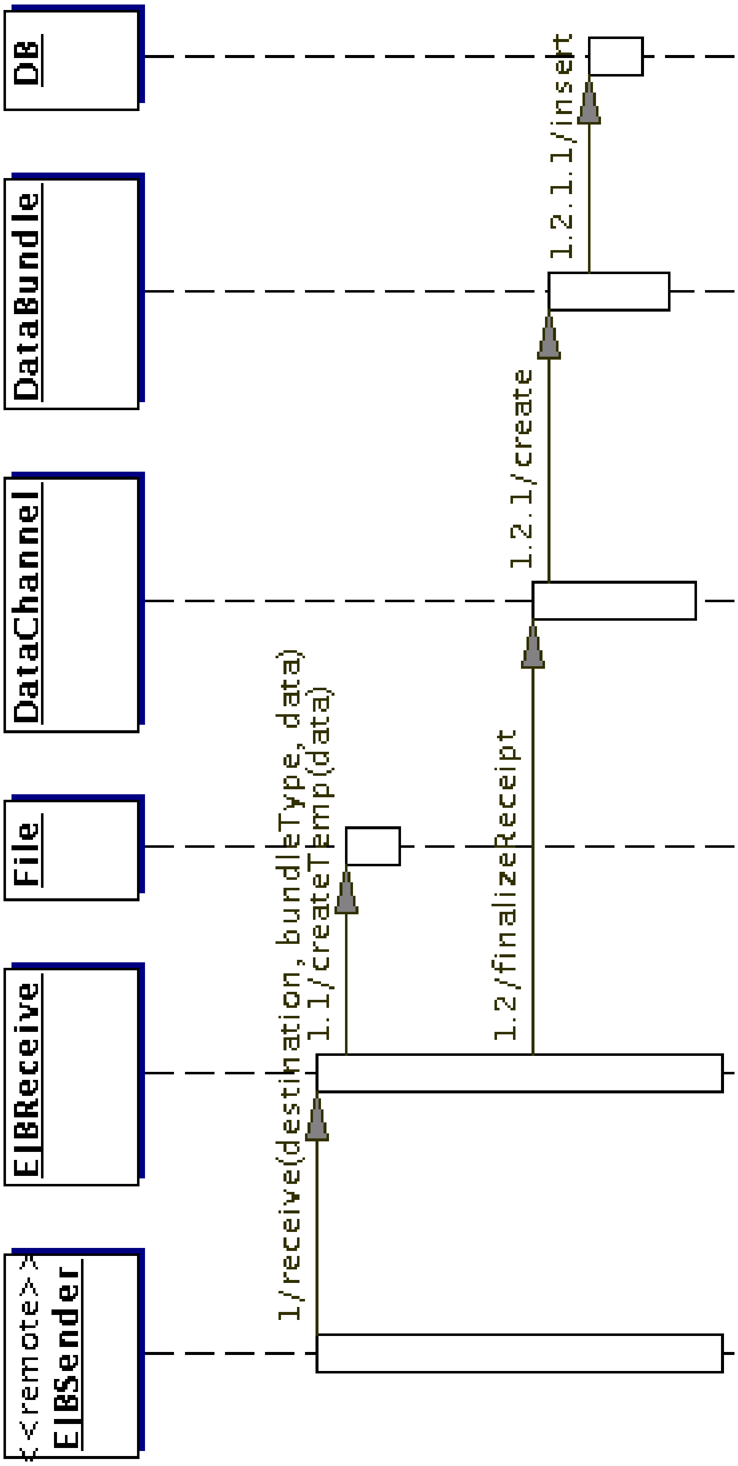

Figure 6.9: Receiving Data Sequence Diagram

of those listed in Section 6.4.3.

In message 1.1 the *EJBReceiver* creates a temporary file, and saves all of the bundle data into it. It then notifies *DataChannel* that a new bundle has arrived with message 1.2. At this point, *DataChannel* creates a new *DataBundle* object which stores the destination, the bundle-type, and the path to the temporary file that contains the bundle data. When it is created, the *DataBundle* object inserts an entry into the local database with all of its attributes and the timestamp of when it was created. The timestamp is used to make sure that bundles are processed in the proper order when they are pulled off the queue by the application.

Data bundles wait on the queue until they are pulled off by the application that they were intended for. Message 1.1 in Figure 6.7 shows *Dirsync* using the `DataChannel.getBundles` method to query any waiting messages from the queue. Just before a data bundle is passed to the requesting component, it is marked in the database as handled by the DataChannel component, making it impossible for a single data bundle to be handled more than once.

6.5 Summary of "Real-World" Features

While the functionality of the Dirsync system is fairly simple, it exhibits enough complexity to provide a solid foundation for studying the different types of reconfigurations that might be found in typical operational systems.

Some of the specific features of Dirsync that are representative of many real-world business applications are as follows:

(1) *Client-server and peer-to-peer relationships* Typically, client-server architectures are easier to reconfigure because there is a clear hierarchy of nodes, and a proper ordering for changes is usually straightforward to determine. With peer-to-peer architectures, certain reconfigurations need to occur on both nodes

at once.

(2) *It is a service that is always running.* This is a very common characteristic of multi-user distributed systems; there must always be some software that is ready to accept requests. This presents a reconfiguration challenge because changes must be made without disrupting the service too much.

(3) *Network topology is dictated by application settings.* A dynamic network topology is fairly common, and presents a configuration challenge since inter-node dependencies can only be determined based on the current configuration. This also means that some changes to the applications parameters will require new bindings to be made between remote components.

(4) *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 configuration challenge, because changes must be coordinated between an external software application and the software components of the system.

6.6 Dirsync System Configurations

One of the major motivations behind developing the Dirsync application was to have a non-trivial system to use as a basis for reasoning about its likely configurations. The following sections outline a potential evolution of a Dirsync installation. The changes that are highlighted were chosen to emphasize different changes that commonly occur with component-based distributed systems.

6.6.1 Pre-Installation Assumptions

The following assumptions are made about the baseline configuration available on each node:

Release	**Summary of Changes**
1.0	Initial installation. Sharing `/dir1` with a master-slave relationship between Node 1 and Node 2.
2.0	The synchronization directory is changed on Node 2.
3.0	`/dir2` is added as a new directory to synchronize. This directory is shared in a peer-to-peer fashion with Node 1 being the peer-master.
4.0	File change algorithm is enhanced to include a checksum of the contents of the file.
5.0	The format of the command file is changed to include the last modified time.
6.0	Convenience method is added to the PropertyMgr component to allow for easier access to integer properties.
7.0	The EJBReceiver component (a sub-component of the DataChannel component) is enhanced to allow for more efficient data transmission.

Table 6.3: Dirsync System Evolution

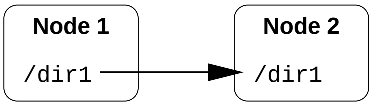

Figure 6.10: Initial Dirsync Topology

- Each node that needs to have Dirsync installed has a properly configured J2EE server.
- J2EE server provides a database connection to a local database.
- Local database does not have any objects (tables, sequences, etc...) created that will conflict with the objects needed by the components in the Dirsync system.
- The user that is connected to the database has permissions to create objects.
- The J2EE server supports dynamic deployment of EJBs.

6.6.2 Initial Install

The initial Dirsync network topology for this discussion is depicted in Figure 6.10. The components must be installed on both nodes with deployment descriptors that allow Node 1 to invoke the necessary data channel components on Node 2.

The following steps must be performed on both Node 1 and Node 2.

(1) The database objects are created:

  - `LOG_ENTRY` - The table that the Logging component stores messages in.
  - `LOG_ENTRY_SEQ` - The sequence that is used to generate numeric primary keys for each of the log messages.
  - `PROPERTY` - The table that stores the name-value property pairs.
  - `FILE_ENTRY` - The table that stores the state information for each file that is under Dirsync's control.

- `DATA_BUNDLE` - This table stores incoming data packages temporarily until they have been handled by their intended recipient.

(2) Initial property values are inserted into the PROPERTY table.

(3) Component artifacts must be moved into a location on each machine from which they will be deployed by the J2EE server. NOTE: The contents of the artifacts have to be customized for each machine since the deployment descriptor includes the network location for remote EJB references.

(4) The `activate` method on the Dirsync component is invoked.

6.6.3 Change Node 2 Directory Location

In this reconfiguration, none of the software artifacts need to be changed, just one of the parameters that is used by the DirsyncSys component. After this reconfiguration, the target directory on the file system of Node 2 will be different.

The following steps must be performed on Node 2:

(1) The property that is used to determine the actual file system location for `dir1` on Node 2 is changed to its new value.

(2) The DirsyncSys component must be notified to reload its settings from the database. One way to accomplish this is to redeploy the DirsyncSys component.

This is an example of a single node parametric change.

6.6.4 Add New Directory to be Synchronized

Once this reconfiguration has been performed, the network topology will be as it is depicted in Figure 6.11. `dir1` will stay as it is, but a new peer-to-peer directory will be scheduled for synchronization. When one Dirsync node connects to another to perform the synchronization of a particular directory, one of the first things they do

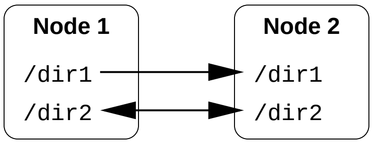

Figure 6.11: Final Dirsync Topology

is verify that their configuration settings for the directory in question are consistent. For this reason, the timing of the following changes is important. Since Node 2 is the peer-slave, the changes must be made to its configuration before Node 1 is reconfigured.

(1) Properties related to `dir2` need to be added to the `PROPERTIES` table in the database.

(2) The DirsyncSys component needs to be forced to reload its properties. This can be accomplished by re-deploying the component.

(3) A new timer needs to be added (Node 1 only since this is the peer-master).

This is an example of a multi-node parametric reconfiguration.

6.6.5 Enhance File Status Algorithm

In order to better determine the status of a file on the file system, the DirState component needs to be re-implemented to compute and store a checksum of the contents of each file that is being synchronized.

This reconfiguration only affects the algorithm that is used to determine if a particular file needs to be synchronized, so the timing of the reconfiguration on each of the nodes is unimportant.

The following steps need to be taken:

(1) Alter the structure of the `FILE_ENTRY` table by adding a new field to store the checksum value.

(2) Deploy a new version of the DirState component that handles the calculation and storage of the checksum value.

(3) Redeploy the Dirsync component to make sure it is bound to the most recent implementation of the DirState component.

This is an example of a single node implementation reconfiguration.

6.6.6 Change Command Format

When a file is found to be out of date, Dirsync sends its contents to the appropriate slave or peer. To make a more accurate replication, the file modification time should also be maintained during a synchronization. To handle this, the format of the file-data bundle is enhanced to include the modification time as well as the file contents. Since the format of data is changed by this modification, it is critical that the reconfiguration is coordinated across both machines.

The following steps need to be taken:

(1) Deploy new version of the Dirsync component with the enhanced data format.

This is an example of a multi-node implementation reconfiguration.

6.6.7 Add Convenience Method for Integer Properties

The initial version of the Property component only provided a means of getting the values of properties as Strings. To provide a more usable interface and consistent error checking for properties, the Property component was enhanced with a method for retrieving Integer properties. The Dirsync component uses an integer property to determine the synchronization period for each of its directories.

Since this change only effects each node locally, the reconfiguration of each node does not need to be coordinated.

The following steps need to be taken:

(1) Deploy new version of Property component with enhanced interface.

(2) Deploy new version of Dirsync component with an updated implementation that takes advantage of the new interface.

This is an example of a single node interface reconfiguration.

6.6.8 Change Interface of EJBReceiver

The DataChannel component is used to send data from one node to another. To help facilitate different communication types, the DataChannel uses pairs of plug-ins to do the actual sending and receiving of data. In the initial installation, data is sent using the EJB scheme where there is an EJB that can receive data one byte at a time. To improve efficiency, this interface is changed so the EJB can receive an entire array of bytes with a single method call. Since this effects the way data is transmitted between the nodes, the reconfiguration must be coordinated carefully.

The following steps need to be taken:

(1) Deploy new version of DataChannel component that has the new EJBReceiver interface and an updated version of the EJBSender implementation that conforms to the new EJBReceiver interface.

This is an example of a multi-node interface reconfiguration.

Chapter 7

`EJB-ARK`

As part of this project, deployment management software, known as the EJB Automatic Reconfiguration frameworK (`EJB-ARK`) was developed. It is a software system designed to facilitate the management and automation of the software deployment life cycle for EJBs. `EJB-ARK` focuses on managing the software consumer deployment life cycle activities: **install**, **activate**, **update**, **reconfigure**, **adapt**, **deactivate**, and **remove**. By helping to manage the software consumer side of the the deployment life cycle in an automatic fashion, it is hoped that distributed systems will be able to be evolved and reconfigured in a much stable fashion with less downtime.

`EJB-ARK` provides some standard functions like the ability to download component packages over the Internet, and load them into the EJB container. Some aspects of the framework manipulate the component package descriptors to provide much more granular control over the EJB system, down to the individual bindings between components. In this way, the commands that `EJB-ARK` makes available to the user define the **assembly language** of EJB-based deployment management.

`EJB-ARK` is a management framework only, 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 sanity checking, to make sure the user does not put the system into an unstable state unwittingly. However, in most cases `EJB-ARK` allows the user to force a particular action, thereby allowing the user to fully control the configuration of the system as they see fit.

7.1 Overview

EJB-ARK is a tool designed to provide management and automation of the software deployment life cycle for EJBs. The software deployment life cycle and the EJB-specific actions that comprise it are described elsewhere in this report. EJB-ARK is capable of performing all of the reconfiguration types outlined in Section 5 with a minimum of disruption to the system.

EJB-ARK has the following distinct parts which are illustrated in Figure 7.1:

(1) Application Server Module (ASM) – This software works in cooperation an EJB 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 ASM is the piece of the framework that represents the software consumer site.

(2) Workbench – This application provides the administrator of the system with a tool for controlling the ASMs directly. The Workbench also allows users to execute "scripts" of deployment actions with the workbench. The Workbench provides a mechanism to bridge between the Repository which is controlled by software producers, and the ASMs which represent the software consumers.

(3) Repository – The repository is 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 EJB-ARK system; there is no specialized software running at the Repository. In practice the Repository can be any file server that is available to all of the ASM instances. This represents the location to which a software producer would release software components.

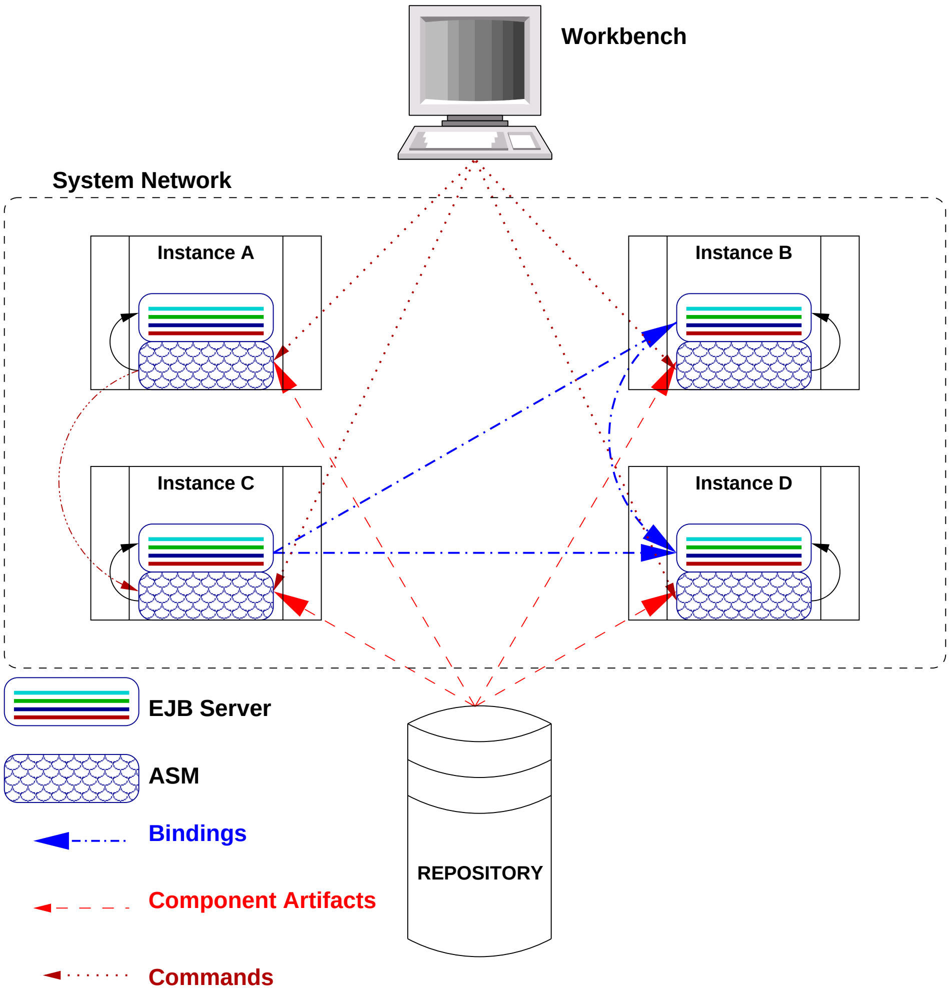

Figure 7.1: `EJB-ARK` System Overview

Changes are made to a system by an end user (most likely an administrator or operator) through the Workbench. The Workbench provides the user with the capability of issuing `EJB-ARK` instructions one at a time, or running a script of commands. If multiple instructions (on one or more ASM instances) need to be performed within the same transaction they need to be executed from within a script. Most of the commands are depicted in Figure 7.1 as originating from the Workbench; a few go between ASM instances. These inter-instance commands are needed to ensure that any changes to bindings are approved by both the client and the server before they are completed.

The programatic interface into the `EJB-ARK` system that the Workbench uses is also available to other applications. The main programatic interface into the ASM is through a Connection. Instances of the Connection class provide strongly typed methods for executing commands on the ASM that they are associated with. In order to get proper transactional processing, some care must be taken when using the Connection. To illustrate this, the steps that the scripting engine takes are listed below:

(1) *Retrieve Connection from primary instance* – If multiple instances are involved in the execution of a script, one of them needs to be chosen as the primary instance. This can be done arbitrarily, there is no special processing that has to be done as a result of this.

(2) *Retrieve Connections for other instances* – The primary Connection is used to open connections onto the other instances. These new connections are all considered to be in the same transaction context as the primary Connection.

(3) *Issue Commands* – With the proper Connection, issue commands to any of the instances.

(4) *Commit or Rollback* – Using the primary Connection object, the entire transaction can be committed or rolled back.

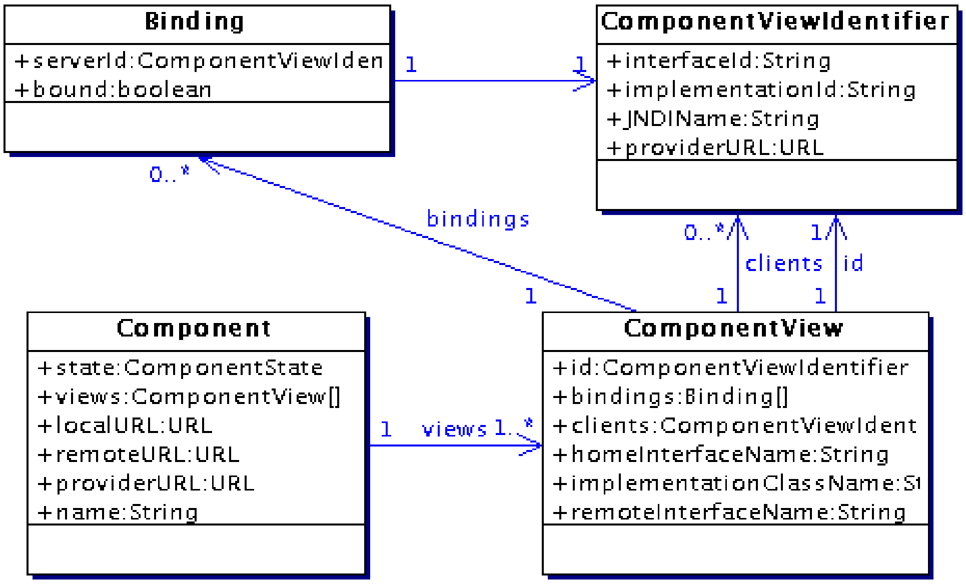

Figure 7.2: Domain Model

Section 7.3 describes EJB-ARK's handling of transactions in more detail.

7.2 Domain Model

Before dealing with the details of the EJB-ARK framework, it is important to define the domain language that it uses to reason about the software system. Figure 7.2 depicts the important domain classes and their relationships.

- Component – The component is the fundamental unit of software that EJB-ARK manages. Conceptually, a component is a self-contained module of a system that must be treated as a unit. In practice a component is represented by a single EJB package, and it is loaded and unloaded from the EJB server as a unit. In EJB-ARK a Component is identified by the combination of its name and its remoteURL. Typically the remoteURL will point to the Repository that is being used by the system. This naming convention assumes that any new software released by the software producer will not overwrite existing artifacts that are available through the repository. The state of the Component is changed as various commands are issued to the ASM instance that is managing of the Component. Figure 7.4 shows the states that a component moves through as it is managed by an ASM. Other important attributed of a component are the URL of the component package on the local file system, and the URL of the EJB server that they are associated with (through the ASM).

A Component is a collection of one or more Views, the entities that actually provide services.

- ComponentView – As mentioned above, a component provides a well known interface that can be used by its clients. Conceptually, a single component might provide multiple different interfaces for use in different situations. Each of the public interfaces exported by a component is known as a ComponentView. Clients (other ComponentViews) of a Component **bind** to a particular ComponentView that provides an implementation of an interface that they expecting. A ComponentView also contains some EJB specific attributes related to the names of their Home, Remote and Implementation classes. The absolute path to a ComponentView is comprised of the URL of the instance they are running inside, the interfaceId and an implementationId. The interfaceId identifies the exact programatic contract that a ComponentView will fulfill. Since a Component corresponds to a single EJB JAR file, a ComponentView corresponds to a specific EJB that is contained in the EJB JAR file.

  A ComponentView may have zero or more clients bound to it; it also has zero or more bindings to other ComponentViews. A ComponentView is not fully activated until all of its bindings are satisfied. The `EJB-ARK` framework makes sure that a ComponentView will not accept any client bindings until all of its bindings have been satisfied[1] .

[1] Actually, it is possible for the user to force a binding by telling the ASM to ignore the fact that a ComponentView is not fully bound. This is necessary to accommodate circular dependencies between

- ComponentViewIdentifier – As mentioned above, the absolute path to a ComponentView consists of the URL to the EJB provider, the interface identifier (which is a hash of all of the methods of the Home and Remote interfaces of the ComponentView), and an implementation identifier (which is a random identifier that is generated when a component package is first downloaded by an ASM). Using this information it is possible to locate the Home interface of a ComponentView. The ComponentViewIdentifier encapsulates this information, and it is used to identify ComponentViews directly, and also to identify clients and bindings to other ComponentViews. Depending on the situation, some of the attributes of a ComponentViewIdentifier may not be specified.

- Binding – A binding represents a client/server relationship between two ComponentViews. A Binding is managed by the client ComponentView. When it is bound, the ComponentView that will be servicing its requests maintains a list of all the clients that are bound to it. In a binding, the server ComponentView is identified only by its providerURL and interfaceId. When a binding is made, the fully specified ComponentViewIdentifier is stored with the binding, so that it can be unbound successfully. On the server ComponentView the full ComponentViewIdentifiers are stored for each of their clients so that they can be explicitly notified by the framework when they need to freshen their bindings or reload their parameters.

7.3 Transactions

A key feature of `EJB-ARK` is its support for transactional reconfigurations. It is easy to see how a partial reconfiguration could render a software system unusable. `EJB-ARK` supports transactions that are distributed across multiple ASM instances. Currently the Workbench only supports multi-instance transactions during the execution

---
ComponentViews

of scripts. While the Workbench does allow the user to execute individual commands transactionally, any operations that involve transactional changes to multiple instances must be performed through an `EJB-ARK` script.

Although application servers that support EJBs always provide a transaction manager that implements Sun's Java Transaction Service (JTS), `EJB-ARK` manages its own transactions. This was an implementation decision driven by the desire to keep the ASM as clearly separated from the application server that it manages.

The client interface to a `EJB-ARK`, the Connection, supports transactions through the commit and rollback methods. Individual transactions are completely isolated from each other. Optimistic concurrency control is used, which effectively means that the first transaction to modify an object will succeed with subsequent modifications failing with a concurrency violation error.

Figure 7.3 illustrates a distributed `EJB-ARK` transaction. In a distributed `EJB-ARK` transaction, a client (usually the Workbench application) needs to make configuration changes on multiple ASM instances in a single transactional unit. The following requirements exist during `EJB-ARK`'s distributed transactions:

(1) All of the configuration changes must be committed or rolled back as a unit.

(2) None of the configuration changes can be visible to other transactions until they are committed.

(3) Configuration changes made to one instance during the transaction must be visible to the other instances that are involved in the transaction.

To support these requirements, a scheme of parent and child Connections is used. To get proper transaction handling, the client first obtains a Connection to one of the instances involved in the distributed transaction. This Connection is the "parent" for the distributed transaction. When this Connection is obtained, a new, unique transaction identifier is created and associated with the transaction. Connections to other

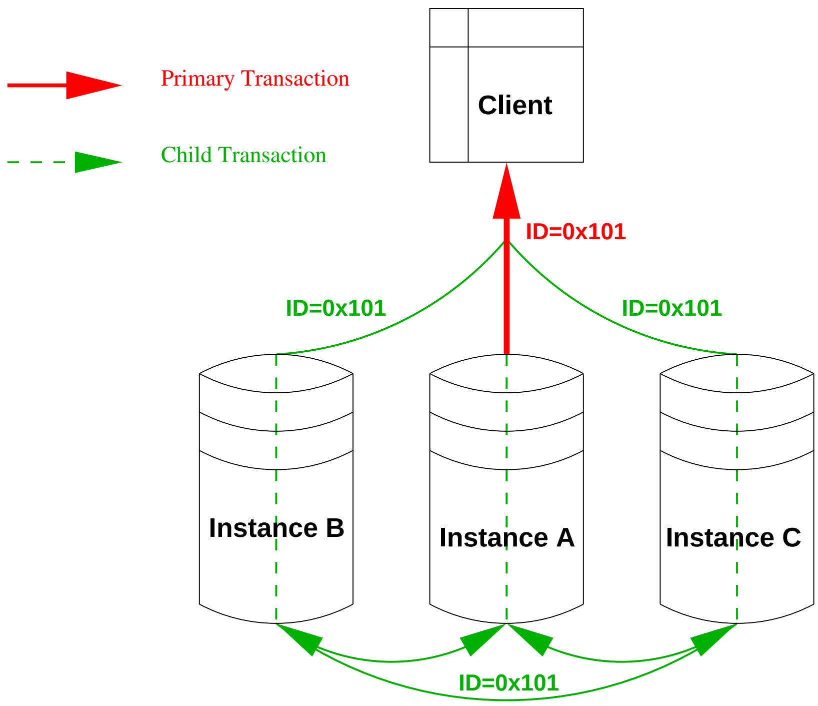

Figure 7.3: `EJB-ARK` Distributed Transaction

"secondary" ASM instances are obtained directly from the parent Connection. When a secondary Connection is created, it inherits the transaction identifier from its parent. When ASM instances communicate directly with each other to fulfill a binding-related request, the transaction identifier is used, so that changes are visible in the proper manner. When the client commits or rolls back the parent Connection, all the child Connections are committed using the standard two-phase commit **prepareToCommit** and **commit** methods.

7.4 Application Server Module

The ASM works in cooperation with a J2EE application server instance to provide management and control over the components that it serves. The division of labor between `EJB-ARK` and the application server is very clear. `EJB-ARK` is responsible for all the processing involved with managing the deployment life cycle of the components including downloading component packages, converting the developer-supplied deployment characteristics into `EJB-ARK`-compliant ones, tracking state and component versions, and managing reconfiguration transactions. In turn, the EJB server is responsible for handling the normal running of the software. `EJB-ARK` only interacts with the application server directly when it deploys or undeploys components. To achieve the necessary granularity for bindings `EJB-ARK` also adjusts the values stored in the naming service (which is typically running as part of the EJB server).

The ASM basically processes instructions that are sent to it through a Connection. These instructions are performed by the user directly or as part of a Workbench script. Instructions are atomic, and effect a single component or binding. The ASM server strictly enforces state transitions for the components that it manages. The following instructions are supported:

(1) `ADD` – This instruction directs an ASM to download a particular component

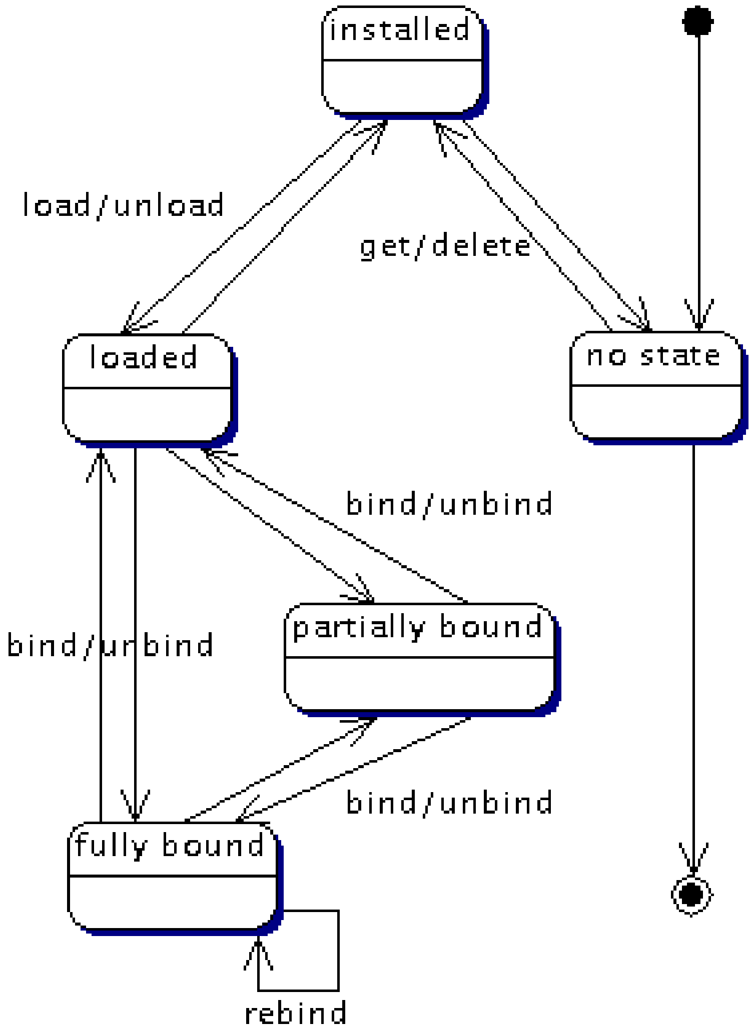

Figure 7.4: Component State Diagram

package from the Repository to the local system. It takes a single argument that is the URL of the component package to download. The component that is released by the software producer is a standard EJB component package. After the package is downloaded, the ASM processes its contents and makes changes to the package descriptor that will allow the component's deployment life cycle to be properly controlled in later instructions. The processing of a component package involves both generic handling of standard component descriptors, and also some processing of the EJB server vendor-specific component descriptors. For this project the open-source EJB server JBoss was used [6]. After the `ADD` instruction has executed, the component will be in the "installed" state. The ASM prevents the same component package from being installed twice.

(2) `DELETE` – This command is the opposite of `ADD`. It takes the identifier of the component to delete as an argument. This instruction directs an ASM to delete the component package from the local system. After a component has been deleted it is in the "no state" state.

(3) `LOAD` – The `LOAD` command directs an ASM to trigger its local application server to put a component into a state where it can make bindings to other components and where it can accept clients. At this point, a standard EJB (not being managed by `EJB-ARK`) would be running and available to accept clients. The implementation of this instruction is necessarily very EJB vendor-specific, since each vendor has their own notion of how to "hot-deploy" component packages.

(4) `UNLOAD` – This instruction is the opposite of the `LOAD` instruction. It takes a component identifier as its argument. This instruction directs an ASM to remove a component from the state where it can make bindings to other components and where it can accept clients. A component cannot be unloaded if it still has active bindings and/or bound clients.

(5) `BIND` – The `BIND` instruction directs an ASM to make a relationship between two ComponentViews. If the components that are supplying the views are on different instances, the ASM that is managing the client component 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 between component views, the `BIND` instruction can force a server ComponentView to accept clients before it is fully bound.

(6) `REBIND` – The `REBIND` instruction is similar to the `BIND` command except the client component view must already be bound to a server component. `REBIND` is used to update the binding of a component view without all of its clients being changed. This instruction is used primarily to change the configuration of a system that is already running.

(7) `UNBIND` – This instruction is the opposite of BIND. It instructs the ASMs to remove a relationship between two component views. The `UNBIND` command cannot be used until all clients of a component view are removed. This command can be forced to remove a relationship in cases of circular dependencies.

(8) `BIND-NAME` – This instruction allows the user to create an arbitrary JNDI alias for any component view. The JNDI names that are chosen by developers are rewritten by the ASM during the processing of the `ADD` instruction. The use of this instruction allows software that is not managed by `EJB-ARK` to retrieve references to a component.

(9) `UNBIND-NAME` – This instruction is the opposite of the `BIND-NAME` instruction, it removes a previously created alias for a component view.

(10) `EXECUTE` – During the processing of the `ADD` instruction, the ASM determines if

a component is executable. Executable components contain at least one ComponentView that implements some predefined interfaces[2] in its Home and Remote interfaces. Once an executable component is in the "fully-bound" state, the ASM can execute it. Executable components can act like scripts that perform a number of deployment actions on the running system. Because the are component like everything else, their actions are transactional and managed completely by the EJB server. Executable components are very useful for making changes to database structure or content.

(11) `RELOAD-PARAMS` – During the processing of the `ADD` instruction, the ASM determines if a component is able to handle the `RELOAD-PARAMS` instruction. Similar to an executable component, a component must have at least one ComponentView that implements some predefined interfaces[3] in its Home and Remote interfaces. Once a reloadable component is in the "fully-bound" state, the ASM can notify it to reload its parameters.

(12) `REFRESH-BINDINGS` – During the processing of the `ADD` instruction, the ASM determines if a component is able to handle the `REFERSH-BINDINGS` instruction. Similar to an executable component, a component must have at least one ComponentView that implements some predefined interfaces[4] in its Home and Remote interfaces. Once a refreshable component is in the "fully-bound" state, the ASM can notify it to refresh its bindings.

7.4.1 Composite Deployment Actions

Using the instructions enumerated in the previous section, very complex deployment actions can be carried out. The instructions provide a granular way of controlling

[2] `ejbark.api.ExecutableHome` and `ejbark.api.ExecutableRemote` respectively.
[3] `ejbark.api.ReloadableHome` and `ejbark.api.ReloadableRemote` respectively.
[4] `ejbark.api.RefreshableHome` and `ejbark.api.RefreshableRemote` respectively.

the deployment of a group of components. The EXECUTE instruction provides a powerful mechanism by which parametric changes can be made to the system by effectively inserting a temporary component into the system that can perform any actions that other components can perform.

By combining the basic instructions in the proper order, complex deployment actions can be performed automatically. These instructions allow the user to control the bindings between components, and also the introduction and removal of components in a safe manner.

7.5 ASM Software Design

The ASM software was designed to run in conjunction with an EJB server. For this project the JBoss EJB server was used [6]. This server is built using Sun Microsystems' JMX framework[5] . The JMX framework allows Manageable Beans (MBeans) to be plugged together easily in a single application and to easily 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 like the naming service (JNDI) 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. During server initiation, the ASM MBean registers a `ConnectionFactory` in the local naming service under the name `java:/EJBARKConnectionFactory`. The Workbench and other ASMs use this factory to obtain instances of the `EJB-ARK` Connection object. Figure 7.5 shows a detailed view of the interactions between the Workbench (or any other client), the ASM and the EJB Server. Figure 7.5 also depicts some components and their bindings.

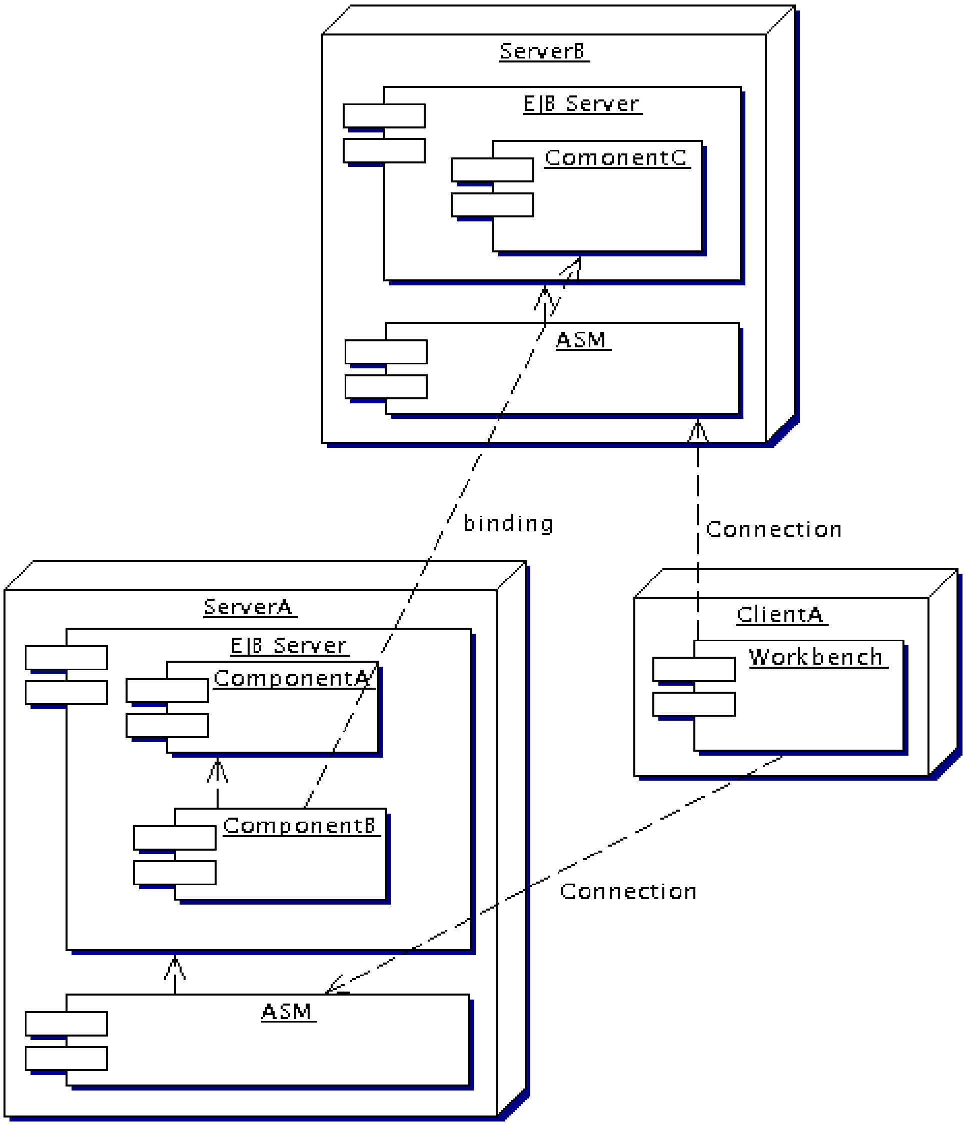

Figure 7.5: Running System Component Diagram

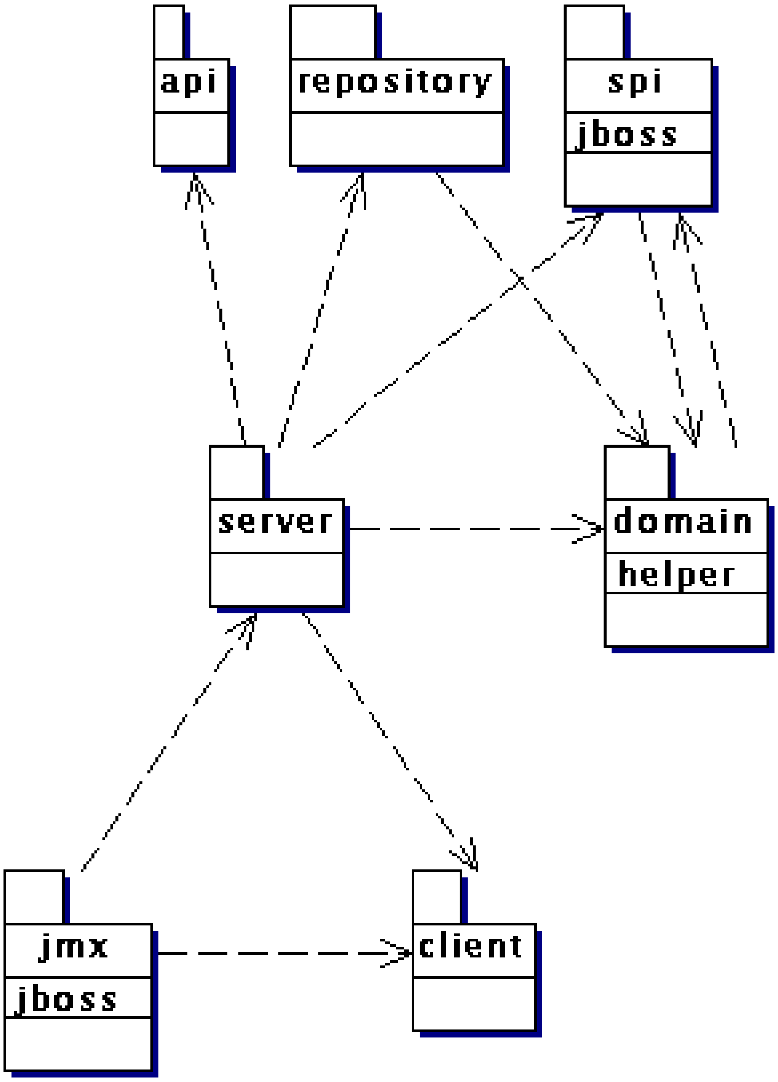

Figure 7.6: ASM Software Packages

7.5.1 ASM Package Hierarchy

The ASM software packages and their dependencies are shown in Figure 7.6. Note: in the software all of the packages depicted in Figure 7.6 are contained in the `ejbark` package. This is elided for clarity.

At the top of the package hierarchy is the `jmx.jboss` package. This contains the classes and interfaces that are required by the JMX framework for an MBean. The functionality of the MBean is very thin, it instantiates a `EJBARKServer` and uses this to create an instance of the `EJBARKConnectionFactory` which it binds to the JNDI naming service with the name `java:/EJBARKConnectionFactory`. The `EJBARKConnection` implementation that is provider by the main server is a remote object that is tied back into the server with a particular transaction id.

The `server` package contains the classes which provide the major processing logic for the ASM. The `EJBARKServer` singleton class is the main entry point for processing instructions. This package also contains implementations of the connection factory and connection objects described in the previous paragraph.

The `domain` package contains implementations of all of the domain objects that were described earlier. The `domain.helper` package contains utility classes for parsing XML descriptors and creating instances of the major domain objects. Some of the XML descriptors that need to be parsed are specific to the EJB server, so the helpers delegate some of the parsing to the registered service provider class, in this case it is JBoss specific.

The EJB server-specific processing is all abstracted by the `J2EEServiceProvider` interface which is in the `spi` (Service Provider Interface) package. This package also has a factory for obtaining the correct `J2EEServiceProvider` instance for the current setup. `spi.jboss` contains the `J2EEServiceProvider` implementation specific to JBoss. In addition to provide methods to help parse EJB server specific descriptors, the service

[5] Other EJB servers, such as BEA's WebLogic, are also built on the JMX framework

provider instance also provides the specific implementation for loading and unloading a component package into the EJB container.

The `api` package contains interfaces that need to be implemented by ComponentViews to enable special processing. These include the `ExecutableComponentView` remote interface, and `ExecutableComponentViewHome` which is a home interface. For a ComponentView to be considered "executable," it must have these interfaces serve as its remote and home interfaces.

The `repository` package contains a transactional repository for domain objects that are managed by the ASM. The `Repository` singleton object manages the separation of transaction contexts and provides support for managing proper concurrency of domain objects between transaction contexts.

7.5.2 `EJB-ARK` and JNDI

Most EJB servers use Sun Microsystems' Java Naming and Directory Interface (JNDI) as a mechanism to make instances of Home interfaces available to a component's clients. Typically, developers include the string name that a component will be bound to in the local JNDI service. Clients of the component know this name, and use it to do their lookups. This arbitrary name poses a few problems when maintaining components in a server:

(1) Clients have this name hard coded somewhere in their software or descriptors (for EJB clients). 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.

(2) JNDI only allows one object to be bound to a particular name, so new versions of a component force prior versions to be unloaded.

Developer	EJB-ARK
componenta/ComponentA	ejbark/19887402/-199480
componentb/ComponentB	ejbark/7730183/128765309

Table 7.1: Remapping of JNDI Names

The primary mechanism that EJB-ARK uses to achieve tighter control over the bindings between components is to manipulate the JNDI bindings after a component has been loaded by a server. The following list details the ways JNDI is used by EJB-ARK to control bindings:

- **Component JNDI Name**

  Before a component is loaded into the server, the name chosen by the developer for each of its views are changed to be a unique identifier for the particular component view. To achieve this, the name includes a hash of the external interface of the component view and a random number. The interface hash is calculated by combining all the methods supplied by the home and remove interfaces of the component view, this groups component views that export the same methods. The random number is used to differentiate all of the component views that implement the same interface. Table 7.1 shows an example of a remapping of component JNDI name. In this example the developer chose "component/ComponentA" as the JNDI name for this component, EJB-ARK computed the hash of the external interface of the component view to be 19887402 and the random number that was generated was -199480. All EJB-ARK generated JNDI names begin with "ejbark".

- **Binding JNDI Names**

  Before a component is loaded into the server, the JNDI name of each of the external bindings is changed to be a unique identifier for the binding. When developers write components, they decide on the name that they will use in

Software	JNDI	Bound Object
ejb/ComponentA	componenta/ComponentA	componenta.ComponentAHome
ejb/ComponentA	ejbark/7730183/128765309/19887402	UNBOUND

Table 7.2: Binding Setup from ComponentB to ComponentA

their software to refer to a particular component. Using the package descriptors, this name is bound to the actual JNDI name of the component that is being used. For each component that is referred to this way `EJB-ARK` replaces the JNDI name in the descriptors with a name that includes the unique name of the client component (as described above) and the interface hash of the server component.

Table 7.2 shows the differences between the way the developer sets up a binding from ComponentB to ComponentA (first row) and the changes that `EJB-ARK` makes before passing the component package to the EJB server. The first column is the string that is used in the ComponentB implementation, this is not changed by `EJB-ARK`. The second column is the JNDI name that the software string is bound to. Note that the developer includes the explicit JNDI name shown in Table 7.1 for ComponentA. The JNDI name generated by `EJB-ARK` is similar to the unique name generated for ComponentB except the interface for ComponentA is appended, this represents *the binding from this version of ComponentB to any version of ComponentA that has the same interface*. The third column shows the object that is initially bound to the JNDI name from column 2. With the developer setup, the home interface of ComponentA is bound to the name. With `EJB-ARK` managing the bindings, there is nothing bound to the JNDI name initially.

- **Binding Creation**

  When a binding is created, `EJB-ARK` attaches a link reference (like a soft link

Name	Bound Object
`ejbark/7730183/128765309/19887402`	UNBOUND
`ejbark/7730183/128765309/19887402`	`link -> ejbark/19887402/-199480`

Table 7.3: Binding Creation from ComponentB to ComponentA

in UNIX file systems) to the binding identifier in the JNDI tree of the client component. This link actually forwards to the unique identifier of the server. When the client component looks up the references to the server component, the local name it uses in software is forwarded to the unique binding name, which is forwarded to the unique name for the server component that it is bound to.

Table 7.3 shows the change that is made when a binding is created by `EJB-ARK`. Initially the unique binding name is unbound in the JNDI context (first row). When `EJB-ARK` creates a binding, it binds a link to the unique identifier of the server component into the binding name.

This JNDI name rewriting is handled automatically for all components that are managed using `EJB-ARK`. One side effect of this is that any software that is a client of one of the components managed by `EJB-ARK`, but that is not itself managed by `EJB-ARK` does not have any way of binding to a component directly. To help alleviate this, the `BIND-NAME` instruction can be used to create an alias for a particular component view with any name. So, if there was a stand-alone application that needed to use ComponentB, and it was expecting ComponentB's Home interface to be bound to the JNDI name "componentb/ComponentB", the `BIND-NAME` instruction could be used to create an alias from "componentb/ComponentB" to ComponentB's unique name $ejbark/7730183/12876309$.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE ejbark-collection SYSTEM "ejbark-collection.dtd" >
<ejbark-collection name="SampleCollection">
    <component>ComponentAB/InitialInstall/ComponentA.jar</component>
    <script>ComponentAB/InitialInstall.xml</script>
    <component>ComponentAB/InitialInstall/ComponentB.jar</component>
    <component>ComponentAB/ImplUpgrade/ComponentA.jar</component>
    <script>ComponentAB/ImplUpgrade.xml</script>
    <collection>Dirsync.xml</collection>
</ejbark-collection>
```

Figure 7.7: EJB-ARK Collection Descriptor

7.6 Repository

The repository is a conceptual entity in the EJB-ARK framework that is able to provide Component archives on request. The role of Repository can be filled by any file server that is reachable from all nodes in the system when the Component archive is loaded onto a particular ASM. The EJB-ARK does not require any specialized software to be running on the Repository. The Repository represents the location that software producers **release** their software to, and also where software is retired from.

In addition to component packages, two other types of artifacts are stored in the Repository: collections and scripts. Both of these artifacts are used by the Workbench.

Collections are groups of component packages, scripts, and other collections that provide software producers with a way of logically grouping the files that are needed by software consumers. Using collections, artifacts can be grouped together in a way that makes it easy for the user of the Workbench to operate on them. A collection is an XML-based description of all of the items that it represents. Figure 7.7 contains a sample collection descriptor. For each item in the collection the descriptor contains the relative path in the Repository from the location of the collection descriptor to the item.

Scripts are also XML descriptions, but they define sequences of EJB-ARK instructions that should be carried out in a single transaction. The script first defines all of the `EJB-ARK` instances that are involved in the script, and then provides the sequence of actions that must be performed on each of the instances.

Figure 7.8 contains a sample `EJB-ARK` script descriptor. In this script, only one host is managed. This 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 server. Step 5 binds the ComponentB ComponentView to the ComponentA ComponentView. The action in step 6 is a special bind instruction that allows the caller to bind an arbitrary name to the specified ComponentView in the naming service of the EJB server; in this case the name "java:/ComponentB" is bound to the ComponentB view. The `bind-default` instruction is needed to allow components or other software that is not managed by the `EJB-ARK` framework to access components. A ComponentView cannot have a default client binding attached to it until all of its bindings are satisfied.

7.7 Workbench

The Workbench is a graphical user interface that allows a user to execute individual instructions to different ASMs. The Workbench also allows the user to download scripts from the Repository and execute them. Figure 7.9 shows the main Workbench user interface. The interface allows the user to manage multiple ASM instances, and multiple repositories.

For each ASM instance that is loaded, all of the components that are known about are listed. For each component that is listed all of its views are shown. Below each view in the tree are all of its bindings. The user can select any of the items in the tree, and details about that item are shown in the right-hand panel. At the bottom of the right hand panel, the instructions that can legally be executed on the selected item are shown. In Figure 7.9 a ComponentView is selected. Its details are listed to

```xml
<?xml version="1.0" standalone="no"?>
<!DOCTYPE ejbark-script SYSTEM "ejbark-script.dtd" >
<ejbark-script name="SampleScript">
    <!-- host declarations -->
    <instance id="moleman" url="moleman:1099" />
    <!-- actions -->
    <get sequence="1" instance-id="moleman" id="compA"
 remote-url="http://repository/ejbark/ComponentA.jar" />
    <get sequence="2" instance-id="moleman" id="compB"
 remote-url="http://repository/ejbark/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" />
    <bind-default sequence="6" instance-id="moleman"
 name="java:/ComponentB"
     server-component-id="compB" server-view-name="ComponentB" />
</ejbark-script>
```

Figure 7.8: EJB-ARK Script Descriptor

Figure 7.9: Workbench User Interface

the right, and the "Bind Default" instruction is available (which means that all of the ComponentView's bindings have been satisfied). The File menu provides a way for the user to add new instances.

The user can also load multiple repositories into the Workbench. For each repository that is loaded, the Workbench lists the items that are available through it. If the item is a script, the user can run it directly by selecting it in the interface, pressing the "Run" button. For component package items, the user can send the package to any of the instances that are already being managed by the Workbench. Once the component package has been retrieved by a particular instance, it can be managed through the instances tab.

Chapter 8

Deployment Management of Dirsync

As a final validation of the `EJB-ARK`, the EJB-based application *Dirsync* was automatically evolved through the series of releases that were outlined in Section 6.6. For simplicity, only two computers were used in the experiment to play all the parts required.

As shown in Figure 8.1, both hosts have a local database running (which is a requirement of Dirsync, not the `EJB-ARK` framework). They also both have the JBoss EJB server running with an attached ASM. Node1 is acting as the Repository server which is implemented by the Apache Web Server. Node1 is also running the Workbench application and driving the experiment.

8.1 Release 1: Initial Installation

The goal of this release is to install the Dirsync software onto two fresh machines and start synchronizing the directories shown in Figure 8.2.

(1) All component packages are retrieved from the repository by each node.

(2) All component packages are loaded into the EJB server.

(3) The database schema is created.

(4) All of the necessary bindings between components are created, this includes bindings between the two nodes for the DataChannel component.

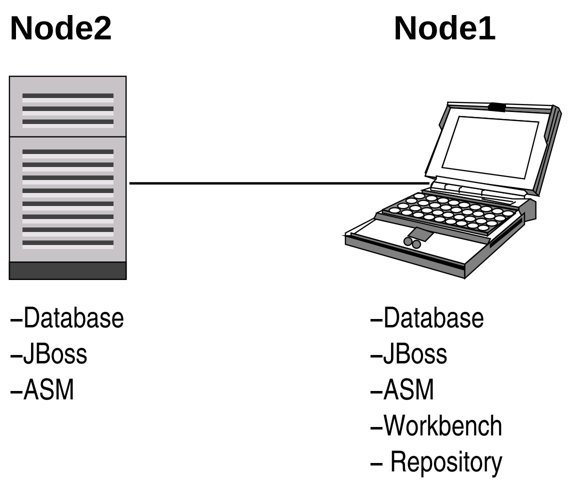

Figure 8.1: Demo Setup

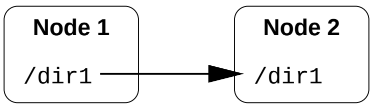

Figure 8.2: Release 1 Topology

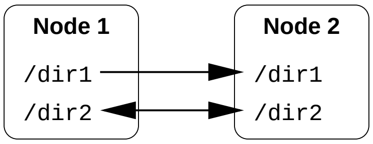

Figure 8.3: Release 3 Network Topology

(5) The Dirsync component is executed on each node.

8.2 Release 2: Single Node Parametric Change

The deployment of this release demonstrates a single node parametric change. In it the shared directory `/dirsync/dir1` is changed to `/dirsyc/new-dir1`.

(1) ADD, LOAD, BIND, and EXECUTE a temporary executable component that performs the following tasks:

   (a) Dirsync component is stopped.

   (b) The directory is renamed

   (c) Dirsync properties are updated.

(2) RELOAD-PARAMS on DirsyncSys component.

(3) Execute Dirsync.

8.3 Release 3: Multiple Node Parametric Change

In this release, a second directory is shared between Node1 and Node2 in a peer-to-peer fashion with Node1 being the **peer-master**.

(1) ADD/LOAD/BIND/EXECUTE a temporary executable component that performs the following tasks (on both Node2 and Node1).

   (a) Stop Dirsync.

(b) Add Dirsync properties

(2) RELOAD-PARAMS on DirsyncSys component on each node.

(3) Execute Dirsync on each node.

8.4 Release 4: Single Node Implementation Change

In this release, the algorithm used to determine if a file has changed on the local file system is updated to include a checksum (previously it used modification time and size).

(1) ADD, LOAD, BIND and EXECUTE a temporary executable component that alters the database schema to include the CHECKSUM field.

(2) ADD/LOAD/BIND a new version of the DirState component with the updated algorithm.

(3) REBIND Dirsync component to the new DirState component.

8.5 Release 5: Multiple Node Implementation Change

In this release, the format of the data packages that are used to send the contents of a changed file to another node is enhanced to include the last modification time of the file. This allows the receiving node to have a more faithful synchronized version.

(1) ADD, LOAD, BIND and EXECUTE a temporary executable component that stops Dirsync (on each node).

(2) UNBIND, UNLOAD, and DELETE existing Dirsync release.

(3) ADD, LOAD, BIND, and EXECUTE new Dirsync component.

8.6 Release 6: Single Node Interface Change

In this release, a new version of the PropertyMgr component is installed that has an additional property access method that converts String to an integer automatically. A new version of DirsyncSys component is installed that uses this new interface method. NOTE: the previous version of the PropertyMgr is kept with all of its other clients.

(1) ADD, LOAD and BIND new PropertyMgr component.

(2) ADD, LOAD new DirsyncSys component and BIND to the new PropertyMgr component.

(3) REBIND existing Dirsync component (From Release 5) to the new DirsyncSys component.

8.7 Release 7: Multiple Node Interface Change

In this node the low-level interface to the DataChannel component is updated to allow for more efficient data transport between instances.

(1) ADD, LOAD new DataChannel component. Also BIND it to its local services. This happens on both nodes.

(2) BIND DataChannel components to each other across the network.

(3) REBIND Dirsync component to new DataChannel components.

8.8 Performance Numbers

The following data were collected over several runs of the complete Dirsync system evolution on nodes 1 and 2. Node2 is a Pentium III 300 with 164 MB RAM running Red Hat Linux 7.0. Node1 is an AMD K-6 435 with 128MB RAM also running Red

Instruction	Average	Std. Dev.	Median	Sample Count
ADD	1368	1044	950.5	56
BIND	80	160	28	86
DELETE	5	6	3	16
LOAD	2866	1681	2349	56
REBIND	85	43	72	8
UNBIND	44	30	32.5	28
UNLOAD	252	203	194	16

Table 8.1: Instruction Timing Results (ms)

Hat Linux 7.0. Both machines were using Sun Microsystems' JDK version 1.3 with the hotspot compiler.

Note that the instructions `EXECUTE`, `RELOAD-PARAMS` and `REFRESH-BINDINGS` are not reported in Table 8.1 since these instructions invoke code that is outside the `EJB-ARK` framework and very implementation specific.

The most time consuming instructions were `ADD` and `LOAD`. The timing of `ADD` is driven by network latency issues and repository implementation. The `LOAD` and `UNLOAD` instructions essentially invoke special methods inside the JBoss server implementation to trigger a load / unload of a component package, again the timing of these methods is dependent on code that is outside the JBoss system.

Although `ADD` and `LOAD` both can take a few seconds to complete they are both instructions that can be done outside the critical path of an applications reconfigurations. In other words, component packages can be pre-loaded by the `EJB-ARK` ASM software, and they are incorporated into the current configuration as needed using the `BIND` and `REBIND` instructions which are relatively fast.

Chapter 9

Implications of `EJB-ARK` on Software Producers

As mentioned in Section 1, one of the over-arching goals of this project was to develop a deployment life cycle management framework that provided fine grained control over the EJB deployment life cycle without introducing a lot of added complexity for the software producers.

Section 8 showed that such a framework could be constructed, however, it is important to quantify the additional software development steps that were required to make this work.

In terms of the actual development of the component software and their package descriptors, no customization was needed. The `EJB-ARK` accepts standard EJB component packages as its inputs. This is extremely important because it means that standard EJB components can be managed by `EJB-ARK` framework without modification.

One paradigm shift for software producers would be the development of "executable" components that are only used during the reconfiguration phase. These executable components contain actions that software producers already have to consider when releasing their software, it is just a different way of packaging the actions. Currently, software producers that are releasing EJB components typically include scripts and database DDL for the users to manually execute. Having to package their installation and activation actions as a component would arguably make their job easier since the delivery platform is standardized for them.

The other artifacts that are supported by `EJB-ARK` are quite straightforward to create, and it would be simple to add a software producer module to the Workbench that would allow a producer to automatically generate collection descriptors and also allow a producer to record individual actions into script.

Chapter 10

Conclusion

This project examined the development of the `EJB-ARK`, a management framework for the software deployment life cycle of Enterprise JavaBeans. To understand the ways EJB systems evolve, six classifications of system reconfigurations were created. Additionally, enhancements to the existing EJB component life cycle were identified. A sample EJB system, *Dirsync*, was developed to validate the `EJB-ARK` architecture and design. As a final demonstration the *Dirsync* application was automatically evolved through seven different releases. This study showed that a framework could be developed that would allow a system to be automatically reconfigured with a relatively small impact on the software producers.

Bibliography

[1] Botha, L.J. and J.M. Bishop, *Configuring Distributed Systems in a Java-based Environment*, IEE Proceedings - Software Engineering - Volume 148, Issue 02. April 2001

[2] Mascolo, C., L. Zanolin and W. Emmerich. *XMILE: an XML based Approach for Incremental Code Mobility and Update*, To appear in Automated Software Engineering. Kluwer. 2002.

[3] Hall, R.S., D. Heimbigner and A.L. Wolf *Specifying the Deployable Software Description Format in XML*, SERL Technical Report CU-SERL-207-99, Software Engineering Research Laboratory, Department of Computer Science University of Colorado, March 1999

[4] Hall, R.S., D. Heimbigner and A.L. Wolf, *A Cooperative Approach to Support Software Deployment Using the Software Dock*, Proceedings of the 1999 Int'l Conf. on Software Engineering, Los Angeles, CA, May 1999, pp. 174-183.

[5] Heimbigner D., R.S. Hall and A.L. Wolf, *A Framework for Analyzing Configurations of Deployable Software Systems*, Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), Las Vegas, Nevada, 18-22 October 1999

[6] JBoss Open-Source J2EE Server Web Site, *http://www.jboss.org*

[7] Magee, J., N. Dulay and J. Kramer, *A Constructive Development Environment for Parallel and Distributed Programs*, Proceedings of the 2nd International Workshop on Configurable Distributed Systems, Pittsburgh, 1994.

[8] Monson-Haefel, R., *Enterprise JavaBeans, 2nd Edition*, O'Reilly and Associated, Inc., 2000.

[9] Ng, K. and J. Kramer, *Automated Support for Distributed Software Design*, Proceedings of the 7th International Workshop on Computer-Aided Software Engineering (CASE 1995), Toronto, July 1995.

[10] Ng, K., J. Kramer, J. Magee and N. Dulay, *The Software Architect's Assistant - A Visual Environment for Distributed Programming*, Proceedings of Hawaii International Conference on System Sciences (HICSS-28), January 1995.

[11] Sun Microsystems Developer Web Site,
http://developer.java.sun.com/developer/onlineTraining/EJBIntro/EJBIntro.html

[12] Sun Microsystems Enterprise JavaBean Specification,
http://www.javasoft.com/products/ejb/docs.html

[13] Sun Microsystems Enterprise JavaBean Web Site,
http://www.javasoft.com/products/ejb

[14] Sun Microsystems JMX Web Site,
http://java.sun.com/products/JavaManagement/index.html

[15] van der Hoek, A., R.S. Hall, A. Carzaniga, D. Heimbigner, and A.L. Wolf, *Software Deployment: Extending Configuration Management Support into the Field*, Crosstalk, The Journal of Defense Software Engineering, volume 11, number 2, February 1998