# Java™ 2 Enterprise Edition

# J2EE™ Connector Architecture Specification

**JSR 016**
**Java Community Process**
http://java.sun.com/jcp/

**Specification Lead:**

**Rahul Sharma,**
Senior Staff Engineer,
Sun Microsystems, Inc.

**Technical comments**:
j2ee-connectors-comments@eng.sun.com

**Version 1.0**
**Final Release**


Java Software,
Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94043 U.S.A.
650 960-1300 fax: 650 969-9131

**J2EE (TM) Connectors Specification ("Specification")**

**Version: 1.0**

**Status: Final Release**

**Release: August 22, 2001**



**NOTICE**

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Sun Microsystems, Inc. ("Sun") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this license and the Export Control and General Terms as set forth in Sun's website Legal Terms. By viewing, downloading or otherwise copying the Specification, you agree that you have read, understood, and will comply with all of the terms and conditions set forth herein.

Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun's intellectual property rights that are essential to practice the Specification, to internally practice the Specification solely for the purpose of creating a clean room implementation of the Specification that: (i) includes a complete implementation of the current version of the Specification, without subsetting or supersetting; (ii) implements all of the interfaces and functionality of the Specification, as defined by Sun, without subsetting or supersetting; (iii) includes a complete implementation of any optional components (as defined by Sun in the Specification) which you choose to implement, without subsetting or supersetting; (iv) implements all of the interfaces and functionality of such optional components, without subsetting or supersetting; (v) does not add any additional packages, classes or interfaces to the "java.*" or "javax.*" packages or subpackages (or other packages defined by Sun); (vi) satisfies all testing requirements available from Sun relating to the most recently published version of the Specification six (6) months prior to any release of the clean room implementation or upgrade thereto; (vii) does not derive from any Sun source code or binary code materials; and (viii) does not include any Sun source code or binary code materials without an appropriate and separate license from Sun. The Specification contains the proprietary information of Sun and may only be used in accordance with the license terms set forth herein. This license will terminate immediately without notice from Sun if you fail to comply with any provision of this license. Upon termination or expiration of this license, you must cease use of or destroy the Specification.

**TRADEMARKS**

No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, the Java Coffee Cup logo, and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

**DISCLAIMER OF WARRANTIES**

THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY

PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

**LIMITATION OF LIABILITY**

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or clean room implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license. provided to you under this license.

**RESTRICTED RIGHTS LEGEND**

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

**REPORT**

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your use of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

(LFI#95240/Form ID#011801)

**Table of Contents**

**Table of Contents**

## List of Figures

# 1 Introduction

The Java 2 Platform, Enterprise Edition (J2EE) provides containers for client applications, web components (based on servlets, Java Server Pages) and Enterprise JavaBeans [1] components. These containers provide deployment and runtime support for application components. They provide a federated view of the services provided by underlying application server for the application components.

Containers can run on existing systems; for example, web servers for the web containers; application servers, TP monitors, and database systems for EJB containers. This enables enterprises to leverage both the advantages of their existing systems and those of J2EE. Enterprises can write (or rewrite) new applications using J2EE capabilities and can also encapsulate parts of existing applications in enterprise beans (EJB) or Java Server Pages (JSP).

Enterprise applications access functions and data associated with applications running on Enterprise Information Systems (EIS). Application servers extend their containers and support connectivity to heterogeneous EISs. Enterprise tools and Enterprise Application Integration (EAI) vendors add value by providing tools and frameworks to simplify the EIS integration task.

## 1.1 Overview

The connector architecture defines a standard architecture for connecting the Java 2 Platform, Enterprise Edition (J2EE) platform to heterogeneous EISs. Examples of EISs include ERP, mainframe transaction processing (TP), and database systems.

The connector architecture defines a set of scalable, secure, and transactional mechanisms that enable the integration of EISs with application servers[1] and enterprise applications.

The connector architecture also defines a Common Client Interface (CCI) for EIS access. The CCI defines a client API for interacting with heterogeneous EISs.

The connector architecture enables an EIS vendor to provide a standard resource adapter for its EIS. A resource adapter is a system-level software driver that is used by a Java application to connect to an EIS. The resource adapter plugs into an application server and provides connectivity between the EIS, the application server, and the enterprise application.

An application server vendor extends its system once to support the connector architecture and is then assured of a seamless connectivity to multiple EISs. Likewise, an EIS vendor provides one standard resource adapter and it has the capability to plug in to any application server that supports the connector architecture.

## 1.2 Scope

The version 1.0 of the connector architecture defines:

1.Application server is a generic term used in this document to refer to a middle-tier component server that is compliant to Java 2 platform, Enterprise Edition.

- A standard set of system-level contracts between an application server and EIS. These contracts focus on the important system-level aspects of integration: connection management, transaction management, and security.
- A Common Client Interface (CCI) that defines a client API for interacting with multiple EISs.
- A standard deployment and packaging protocol for resource adapters.

Refer to section 2.2.2 for the rationale behind the Common Client Interface.

## 1.3 Target Audience

The target audience for this specification includes:

- EIS vendors and resource adapter providers
- Application server vendors and container providers
- Enterprise application developers and system integrators
- Enterprise tool and EAI vendors

The system-level contracts between an application server and EIS are targeted towards EIS vendors (or resource adapter providers, if the two roles are different) and application server vendors.

The CCI is targeted primarily towards enterprise tools and EAI vendors.

## 1.4 JDBC and Connector Architecture

The JDBC API defines a standard Java API for accessing relational databases. JDBC provides an API for sending SQL statements to a database and processing the tabular data returned by the database.

The connector architecture is a standard architecture for integrating J2EE applications with EISs that are not relational databases. Each of these EISs provides a native function call API for identifying a function to call, specifying its input data, and processing its output data. The goal of the Common Client Interface (CCI) is to provide an EIS independent API for coding these EIS function calls.

The CCI is targeted at EIS development tools and other sophisticated users of EISs. The CCI provides a way to minimize the EIS specific code required by such tools. Most J2EE developers will access EISs using these tools rather than using CCI directly.

It is expected that many J2EE applications will combine relational database access using JDBC with EIS access using EIS access tools based on CCI.

The connector architecture defines a standard SPI (Service Provider Interface) for integrating the transaction, security and connection management facilities of an application server with those of a transactional resource manager. The JDBC 3.0 specification [3] specifies the relationship of JDBC to the SPI specified in the connector architecture.

## 1.5 Organization

The document starts by describing the rationale and goals for a standard architecture to integrate an application server with multiple heterogeneous EISs. It then describes the key concepts relevant to the connector architecture. This introduction facilitates an understanding of the overall architecture.

The document then describes typical scenarios for the connector architecture. This chapter introduces the various roles and responsibilities involved in the development and deployment of enterprise applications that integrate with multiple EISs.

After forming a descriptive framework for the connector architecture, the document focuses on the prescriptive aspects of the architecture. It introduces the overall connector architecture focussing on the Common Client Interface and system-level contracts.

## 1.6 Document Convention

A regular Palatino font is used for describing the connector architecture.

*A italic font is used for paragraphs that contain descriptive notes providing clarifications.*

It is important to note that the scenarios described in the document are illustrative in scope. The intent of the scenarios is **not** to specify a prescriptive way of implementing a particular contract.

The document uses the EJB component model to describe certain scenarios. The EJB specification [1] provides the latest and most accurate details from the perspective of the EJB component model.

## 1.7 Connector Architecture Expert Group

The following are part of the expert group and have made invaluable contributions to the Connector architecture specification:

- BEA — Pete Homan
- Fujitsu — Yoshi Otagiri, Ivar Alexander
- IBM — Tom Freund, Michael Beisiegel
- Inline — Jack Greenfield
- Inprise — Charlton Barreto
- IPlanet — Tony Pan, Pavan Bhatnagar
- Motorola — Guy Bieber
- Oracle — Dan Coyle
- SAP — Marek Barwicki
- Sun — Rahul Sharma (**Specification Lead**), Fred H. Carter
- Sybase — Rajini Balay, K. Swaminathan
- Tibco — Jon Dart
- Unisys — Lester Lee

## 1.8 Acknowledgements

Shel Finkelstein, Mark Hapner, Vlada Matena, Tony Ng, Bill Shannon and Sekhar Vajjhala (all from Sun Microsystems) have provided invaluable technical input and guidance to the Connector architecture specification. Jean Zeng and Pong Ching also provided useful input to the specification.

Rick Cattell, Shel Finkelstein, Bonnie Kellett and Jeff Jackson have provided huge support to the specification lead in the management of the Connectors expert group.

Tony Ng is leading the effort of providing a reference implementation for the Connector architecture as part of J2EE 1.3 platform. Liz Blair has worked on providing the Compatibility Test Suite (CTS) plan for the Connector architecture.

Beth Stearns provided a great help in doing an editorial review of this document.

# 2 Overview

This chapter introduces key concepts that are required for an understanding of the connector architecture. It lays down a reference framework to facilitate a formal specification of the connector architecture in the subsequent chapters of this document.

## 2.1 Definitions

### Enterprise Information System (EIS)

An EIS provides the information infrastructure for an enterprise. An EIS offers a set of services to its clients. These services are exposed to clients as local and / or remote interfaces. Examples of an EIS include:

- ERP system
- Mainframe transaction processing system
- Legacy database system

There are two aspects of an EIS:

- System level services - for example, JDBC, SAP RFC, CICS ECI
- An application specific interface—for example, the table schema and specific stored procedures, the specific CICS TP program

### Connector Architecture

An architecture for integration of J2EE servers with EISs. There are two parts to this architecture: an EIS vendor-provided resource adapter and an application server that allows this resource adapter to plug in. This architecture defines a set of contracts, such as transactions, security, connection management, that a resource adapter has to support to plug in to an application server.

### EIS Resource

An EIS resource provides EIS-specific functionality to its clients. Examples are:

- A record or set of records in a database system
- A business object in an ERP system
- A transaction program in a transaction processing system

### Resource Manager(RM)

A resource manager manages a set of shared EIS resources. A client requests access to a resource manager to use its managed resources. A transactional resource manager can participate in transactions that are externally controlled and coordinated by a transaction manager.

In the context of the connector architecture, a client of a resource manager can either be a middle-tier application server or a client-tier application. A resource manager is typically in a different address space or on a different machine from the client that accesses it.

This document refers to an EIS as a resource manager when it is mentioned in the context of transaction management. Examples of resource managers are a database system, a mainframe TP system and an ERP system.

### Managed Environment

A managed environment defines an operational environment for a J2EE-based, multi-tier, web-enabled application that accesses EISs. The application consists of one or more application components—EJBs, JSPs, servlets—which are deployed on containers. These containers can be one of the following:

- Web containers that host JSP, servlets, and static HTML pages
- EJB containers that host EJB components
- Application client containers that host standalone application clients

### Non-managed Environment

A non-managed environment defines an operational environment for a two-tier application. An application client directly uses a resource adapter to access the EIS, which defines the second tier for a two-tier application.

### Connection

A connection provides connectivity to a resource manager. It enables an application client to connect to a resource manager, perform transactions, and access services provided by that resource manager. A connection can be either transactional or non-transactional. Examples include a database connection and a SAP R/3 connection.

### Application Component

An application component can be a server-side component, such as an EJB, JSP, or servlet, that is deployed, managed, and executed on an application server. It can also be a component executed on the web-client tier but made available to the web-client by an application server. Examples of the latter type of application component include a Java applet, DHTML page.

### Container

A container is a part of an application server that provides deployment and runtime support for application components. It provides a federated view of the services provided by the underlying application server for the application components. For more details on different types of standard containers, refer to Enterprise JavaBeans (EJB) [1], Java Server Pages (JSP), and Servlets specifications.

## 2.2 Rationale

The following section describes the rationale behind the connector architecture.

### 2.2.1 System Contracts

Currently, none of the existing Java platform specifications address the problem of providing a standard architecture for integration between an application server and EISs. Most EIS vendors and application server vendors use vendor-specific architectures to provide EIS integration.

The connector architecture provides a Java solution to the problem of connectivity between the multitude of application servers and EISs. By using the connector architecture, it is no longer necessary for EIS vendors to customize their product for each application server. An application server vendor who conforms to the connector architecture also does not need to add custom code whenever it wants to extend its application server to support connectivity to a new EIS.

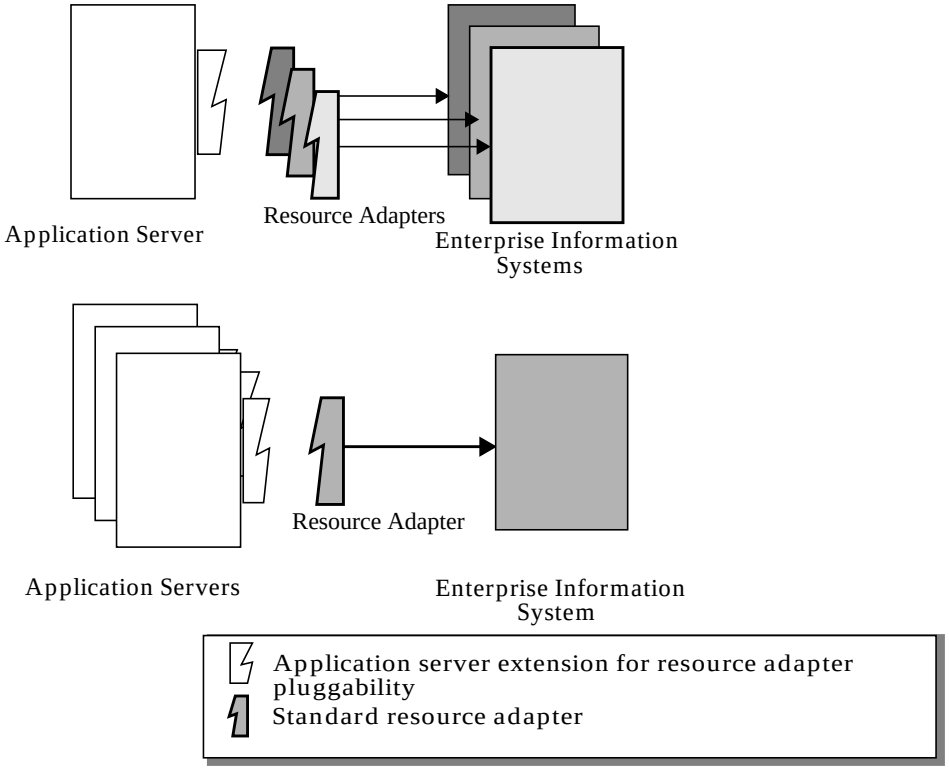
The connector architecture enables an EIS vendor to provide a standard resource adapter for its EIS; the resource adapter plugs into an application server and provides the underlying infrastructure for the integration between an EIS and the application server.

An application server vendor extends its system only once to support the connector architecture and is then assured of connectivity to multiple EISs. Likewise, an EIS vendor provides one standard resource adapter and it has the capability to plug in to any application server that supports the connector architecture.

The following figure shows that a standard EIS resource adapter can plug into multiple application servers. Similarly, multiple resource adapters for different EISs can plug into an application server. This system-level pluggability is made possible through the connector architecture.

If there are *m* application servers and *n* EISs, the connector architecture reduces the scope of the integration problem from an *m x n* problem to an *m* + *n* problem.

**FIGURE 1.0** System Level Pluggability between Application Servers and EISs




## 2.2.2 Common Client Interface

An enterprise tools vendor provides tools that lead to a simple application programming model for EIS access, thereby reducing the effort required in EIS integration. An EAI vendor provides a framework that supports integration across multiple EISs. Both types of vendors need to integrate across heterogeneous EISs.

Each EIS typically has a client API that is specific to the EIS. Examples of EIS client APIs are: RFC for SAP R/3 and ECI for CICS.

An enterprise tools vendor adapts different client APIs for target EISs to a common client API. The adapted API is typically specific to a tools vendor and supports an application programming model common across all EISs. Adapting the API requires significant effort on part of a tools vendor. In this case, the *m x n* integration problem applies to tools vendors.

The connector architecture provides a solution for the m x n integration problem for tools and EAI vendors. The architecture specifies a standard Common Client Interface (CCI) that supports a common client API across heterogeneous EISs.

All EIS resource adapters that support CCI are capable of being plugged into enterprise tools and EAI frameworks in a standard way. A tools vendor need not do any API adaption; the vendor can focus on providing its added value of simplifying EIS integration.

The CCI drastically reduces the effort and learning requirements for tools vendor by narrowing the scope of a *m x n* problem to *m* + *n* problem if there are *m* tools and *n* EISs.

## 2.3 Goals

The connector architecture has been designed with the following goals:

- It simplifies the development of scalable, secure, and transactional resource adapters for a wide range of EISs — ERP systems, database systems, mainframe-based transaction processing systems.
- It is sufficiently general to cover a wide range of heterogeneous EISs. The sufficient generality of the architecture ensures that there are various implementation choices for different resource adapters; each choice is based on the characteristics and mechanisms of an underlying EIS.
- It is not tied to a specific application server implementation but is applicable to all J2EE platform compliant application servers from multiple vendors.
- It provides a standard client API for enterprise tools and EAI vendors. The standard API will be common across heterogeneous EISs.
- It is expressed in a manner that allows an unambiguous determination of whether or not an implementation is compatible.
- It is simple to understand and easy to follow, regardless of whether one is designing a resource adapter for a particular EIS or developing / deploying application components that need to access multiple EISs. This simplicity means the architecture introduces only a few new concepts and places minimal requirements so that it can be leveraged across different integration scenarios and environments.
- It defines contracts and responsibilities for various roles that provide pieces for standard connectivity to an EIS. This enables a standard resource adapter from a EIS vendor to be pluggable across multiple application servers.
- It enables an enterprise application programmer in a non-managed application environment to directly use the resource adapter to access the underlying EIS. This is in addition to a managed access to an EIS with the resource adapter deployed in the middle-tier application server.

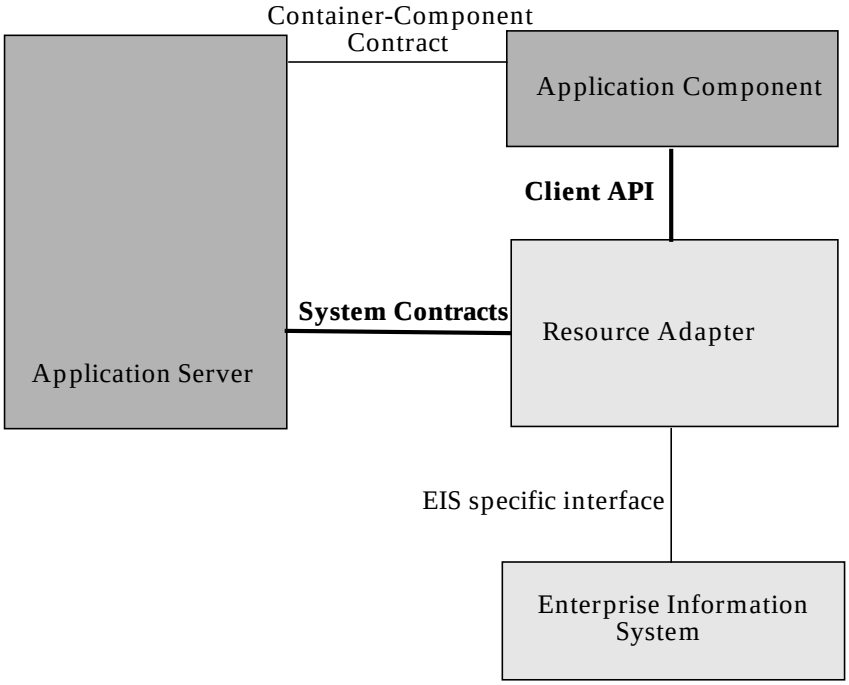
# 3 Connector Architecture

The following chapter specifies an overview of the connector architecture.

Multiple resource adapters—that is, one resource adapter per type of EIS—are pluggable into an application server. This capability enables application components deployed on the application server to access the underlying EISs.

An application server and an EIS collaborate to keep all system-level mechanisms—transactions, security, and connection management—transparent from the application components. As a result, an application component provider focuses on the development of business and presentation logic for its application components and need not get involved in the system-level issues related to EIS integration. This leads to an easier and faster cycle for the development of scalable, secure, and transactional enterprise applications that require connectivity with multiple EISs.

**FIGURE 2.0** Overview of the Connector Architecture




## 3.1 System Contracts

To achieve a standard system-level pluggability between application servers and EISs, the connector architecture defines a standard set of system-level contracts between an application server and EIS. The EIS side of these system-level contracts are implemented in a resource adapter.

A resource adapter is specific to an underlying EIS. It is a system-level software driver that is used by an application server or an application client to connect to an EIS.

A resource adapter plugs into an application server. The resource adapter and application server collaborate to provide the underlying mechanisms—transactions, security, and connection pooling.

A resource adapter is used within the address space of the application server. Examples of resource adapters are:

- A JDBC driver to connect to a relational database (as specified in the JDBC [3] specification)
- A resource adapter to connect to an ERP system
- A resource adapter to connect to a TP system

The connector architecture defines the following set of standard contracts between an application server and EIS:

- A connection management contract that enables an application server to pool connections to an underlying EIS, and enables application components to connect to an EIS. This leads to a scalable application environment that can support a large number of clients requiring access to EISs.
- A transaction management contract between the transaction manager and an EIS that supports transactional access to EIS resource managers. This contract enables an application server to use a transaction manager to manage transactions across multiple resource managers. This contract also supports transactions that are managed internal to an EIS resource manager without the necessity of involving an external transaction manager.
- A security contract that enables a secure access to an EIS. This contract provides support for a secure application environment that reduces security threats to the EIS and protects valuable information resources managed by the EIS.

The Figure 2.0 does not illustrate any contracts that are internal to an application server implementation. The specific mechanisms and contracts within an application server are outside the scope of the connector architecture specification. This specification focuses on the system-level contracts between the application server and EIS.

In the Figure 2.0, the application server and resource adapter are shown as separate entities. This is done to illustrate that there is a logical separation of the respective roles and responsibilities defined for the support of the system level contracts. However, this separation does not imply a physical separation, in terms of an application server and a resource adapter running in separate processes.

## 3.2 Client API

The client API used by application component for EIS access may be defined in terms of:

- The standard Common Client Interface (CCI) as specified in the chapter 9.
- A client API specific to the type of a resource adapter and its underlying EIS. Example of such EIS specific client APIs is JDBC for relational databases.

The Common Client Interface (CCI) defines a common client API for accessing EISs. The CCI is targeted towards Enterprise Application Integration (EAI) and enterprise tools vendors.

## 3.3 Requirements

The connector architecture requires that the connector architecture-compliant resource adapter and the application server support the system contracts. Detailed requirements for each system contract are specified in the later chapters.

The connector architecture recommends (though it does not mandate) that a resource adapter support CCI as the client API. The recommendation enables the connector architecture to provide a solution for the *m x n* integration problem for application development tools and EAI vendors.

The connector architecture allows a resource adapter with an EIS-specific client API to support system contracts and to be capable of standard connector architecture-based pluggability into an application server.

## 3.4 Non-managed Environment

The connector architecture supports access to EISs from non-managed application clients; for example, Java applications and applets.

In a non-managed two-tier application environment, an application client directly uses a resource adapter library. A resource adapter, in this case, exposes its low-level transactions and security APIs to its clients. An application client has to take responsibility for managing security and transactions (and rely on connection pooling if done by the resource adapter internally) by using the low-level APIs exposed by the resource adapter. This model is similar to the way a two-tier JDBC application client accesses a database system in a non-managed environment.

# 4 Roles and Scenarios

This chapter describes a set of roles specific to the connector architecture. The goal of this chapter is to specify contracts that ensure that the end product of each role is compatible with the input product of the other role. Later chapters specify a detailed set of responsibilities for each role relative to the system-level contracts.

## 4.1 Roles

The following section describes roles and responsibilities specific to the connector architecture.

### 4.1.1 Resource Adapter Provider

The resource adapter provider is an expert in the technology related to an EIS and is responsible for providing a resource adapter for an EIS. Since this role is highly EIS specific, an EIS vendor typically provides the resource adapter for its system.

A third party vendor (who is not an EIS vendor) may also provide an EIS resource adapter and its associated set of application development tools. Such a provider typically specializes in writing resource adapters and related tools for a large number of EISs.

### 4.1.2 Application Server Vendor

The application server vendor provides an implementation of a J2EE-compliant application server that provides support for component based enterprise applications. A typical application server vendor is an OS vendor, middleware vendor, or database vendor. The role of an application server vendor is typically the same as that of a container provider.

The J2EE platform specification [8] specifies requirements for a J2EE platform provider.

### 4.1.3 Container Provider

The container provider is responsible for providing a container implementation for a specific type of application component. For example, the container provider may provide a container for EJB components. Each type of application component—EJB, servlet, JSP, applet—has its own set of responsibilities for its container provider. The respective specifications specify these responsibilities.

A container implementation typically provides the following functionality:

- It provides deployed application components with transaction and security management, distribution of clients, scalable management of resources, and other services that are generally required as part of a managed server platform.
- It provides application components with connectivity to an EIS by transparently managing security, resources, and transactions using the system-level contracts with the EIS-specific resource adapter.
- It insulates application components from the specifics of the underlying system-level mechanisms by supporting a simple, standard contract with the application component. Refer to Enterprise JavaBeans specification [1] for more details on the EJB component contract.

The expertise of the container provider is system-level programming, with its focus on the development of a scalable, secure, and transaction-enabled container.

The container provider is also responsible for providing deployment tools necessary for the deployment of application components and resource adapters. It is also required to provide runtime support for the deployed application components.

The container provider typically provides tools that allow the system administrator to monitor and manage a container and application components during runtime.

### 4.1.4 Application Component Provider

In the context of the connector architecture, the application component provider produces an application component that accesses one or more EISs to provide its application functionality.

The application component provider is an application domain expert. In the case of application components targeted towards integration with multiple EISs, various business tasks and entities are implemented based on access to EIS data and functions.

The application component provider typically programs against easy-to-use Java abstractions produced by application development tools. These Java abstractions are based on the Common Client interface (CCI).

The application component provider is not required to be an expert at system level programming. The application component provider does not program transactions, security, concurrency, distribution, but relies on a container to provide these services transparently.

The application component provider is responsible for specifying structural information for an application component and its external dependencies. This information includes, for example, the name and type of the connection factories and security information.

The output of an application component provider is a JAR file that contains the application components and any additional Java classes required for connectivity to EISs.

### 4.1.5 Enterprise Tools Vendors

The application component provider relies on tools to simplify application development and EIS integration. Since programming client access to EIS data and functions is a complex application development task, an application development tool reduces the effort and complexity involved in this task.

Enterprise tools serve different steps in the application development process, as follows:

- Data and function mining tool—enables application component providers to look at the scope and structure of data and functions existing in an EIS.
- Analysis and design tool—enables application component providers to design an application in terms of EIS data and functions.
- Code generation tool—generates Java classes for accessing EIS data and functions. A mapping tool that bridges across two different programming models (object to relational or vice-versa) falls into this category of tools.
- Application composition tool—enables application component providers to compose application components from Java classes generated by a code generation tool. This type of tool typically uses the JavaBeans component model to enhance the ease of programming and composition.
- Deployment tool—used by application component providers and deployers to set transaction, security, and other deployment time requirements.

A number of these tools may be integrated together to form an end-to-end application development environment.

In addition, various tools and middleware vendors offer EAI frameworks that simplify integration across heterogeneous EISs.

### 4.1.6 Application Assembler

The application assembler combines various application components into a larger set of deployable units. The input of the application assembler is one or more JAR files produced by an application component provider and the output is one or more JAR files with a deployment descriptor.

The application assembler is typically a domain expert who assembles application components to produce an enterprise application. To achieve this goal, the application assembler takes application components, possibly from multiple application component providers, and assembles these components.

### 4.1.7 Deployer

The deployer takes one or more deployable units of application components, produced by the application assembler or component provider, and deploys the application components in a target operational environment. An operational environment is comprised of an application server and multiple connected EISs.

The deployer is responsible for resolving all external dependencies declared by the application component provider. For example, the deployer ensures that all connection factories used by the application components are present in an operational environment. To perform its role, the deployer typically uses the application server-provided deployment tools.

The deployer is also responsible for the deployment of resource adapters. Since an operational environment may include multiple EISs, the role of the deployer is more intensive and complex than that in a non-EIS scenario. The deployer has to understand security, transaction, and connection management- related aspects of multiple EISs that are configured in an operational environment.

### 4.1.8 System Administrator

The system administrator is responsible for the configuration and administration of a complete enterprise infrastructure that includes multiple containers and EISs.

In an operational environment that has multiple EISs, the deployer should manage the operational environment by working closely with the system administrators of respective EISs. This enables the deployer to resolve deployment issues while deploying application components and resource adapters in a target operational environment.

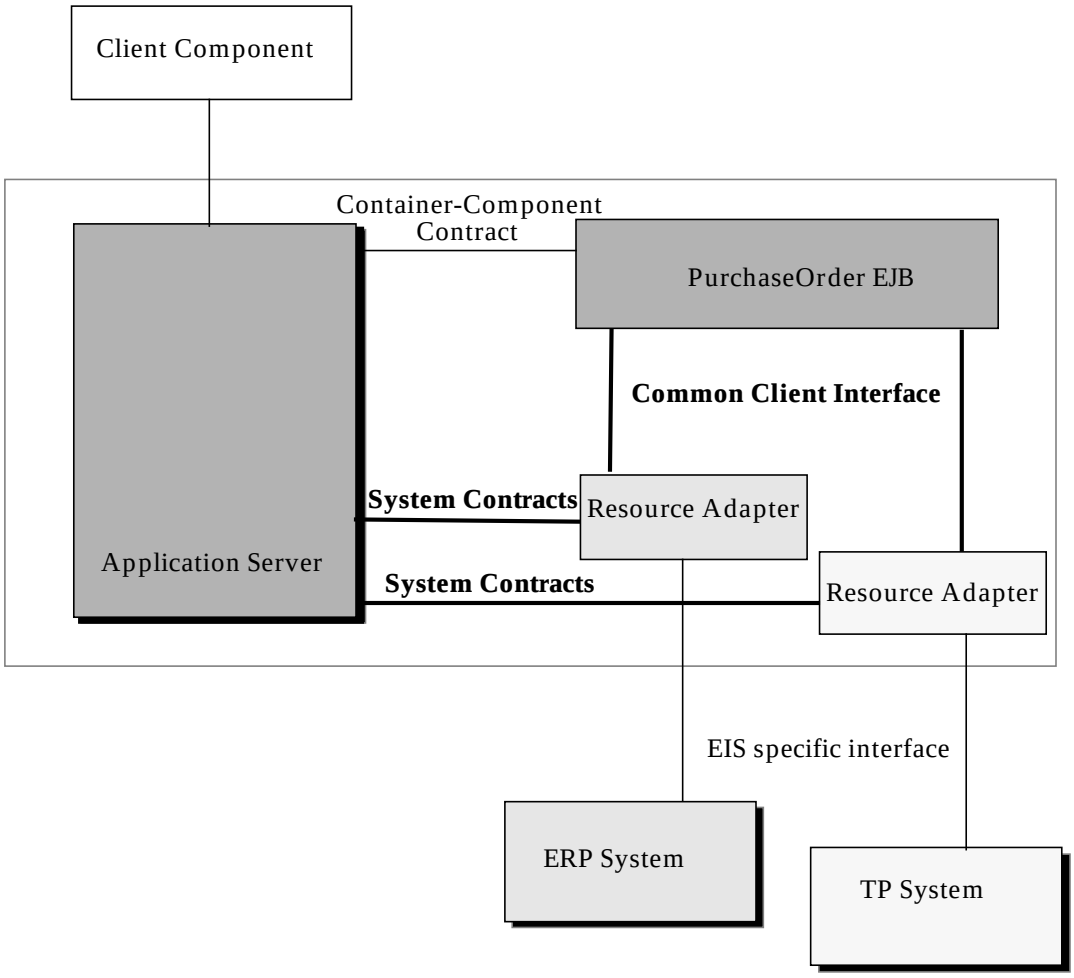
This chapter served as an introduction to the roles involved in the connector architecture. The later chapters specify responsibilities for each role in more detail.

## 4.2 Scenario: Integrated Purchase Order system

This section describes a scenario that illustrates the use of the connector architecture. The following description is kept at a high level. Specific scenarios related to transaction management, security, and connection management are described in subsequent chapters.

The following diagram shows the different pieces that comprise this illustrative scenario:

**FIGURE 3.0** Illustration of an scenario based on the connector architecture




ERP Software Inc. is an enterprise system vendor that provides an enterprise resource planning (ERP) system. ERP Software wants to integrate its ERP system with various application servers. It achieves this goal by providing a standard resource adapter for its ERP system. The resource adapter for ERP system supports the standard transaction, connection management and security contracts. The resource adapter also supports the Common Client Interface (CCI) as its client API.

TPSoft Inc. is another enterprise system vendor that provides a transaction processing (TP) system. TPSoft has also developed a standard resource adapter for its TP system. The resource adapter library supports CCI as part of its implementation.

AppServer Inc. is a system vendor that has an application server product which supports the development and deployment of component-based enterprise applications. This application

server product has an EJB container that provides deployment and runtime support for EJB components. The application server supports the system-level contracts that enable a resource adapter, which also supports these contracts, to plug into the application server and provide connectivity to the underlying EIS. The EJB container insulates EJB components from the transaction, security, and connection management mechanisms required for connecting to the EIS.

Manufacturer Corp. is a big manufacturing firm that uses a purchase order processing system based on the ERP system for its business processes. Recently, Manufacturer has acquired a firm that uses TPSoft's TP system for its purchase order processing. Manufacturer aims to integrate these two systems together into a single integrated purchase order system. It wants a scalable, multi-user secure, transaction-enabled integrated purchase order system that is not tied to a specific computing platform. Manufacturer plans to deploy the middle-tier of this system on the application server from AppServer Inc.

The MIS department of Manufacturer develops a PurchaseOrder EJB that provides an integrated view of the two underlying purchase order systems. While developing PurchaseOrder EJB, the bean provider does not program the transactions, security, or connection management mechanisms required for connectivity to the ERP and TP systems; it relies on the EJB container and application server to provide these services.

The bean provider uses an application programming model based on the CCI to access the business objects and function modules for purchase order processing in the ERP system. The bean provider uses a similar application programming model based on the CCI to access the purchase order processing programs in the TP system.

The MIS department of Manufacturer assembles an integrated web-based purchase order application using PurchaseOrder EJB with other types of application components, such as JSPs and servlets.

The MIS department installs and configures the application server, ERP, and TP system as part of its operational environment. It then deploys the integrated purchase order application on this operational environment. As part of the deployment, the MIS department configures the operational environment based on the deployment requirements for the various application components that have been assembled into the integrated enterprise application.

After deploying and successfully testing the integrated purchase order system, the MIS department makes the system available to other departments for use.

## 4.3 Scenario: Business-to-Business (B2B)

This scenario illustrates the use of the connector architecture in a B2B e-commerce scenario.

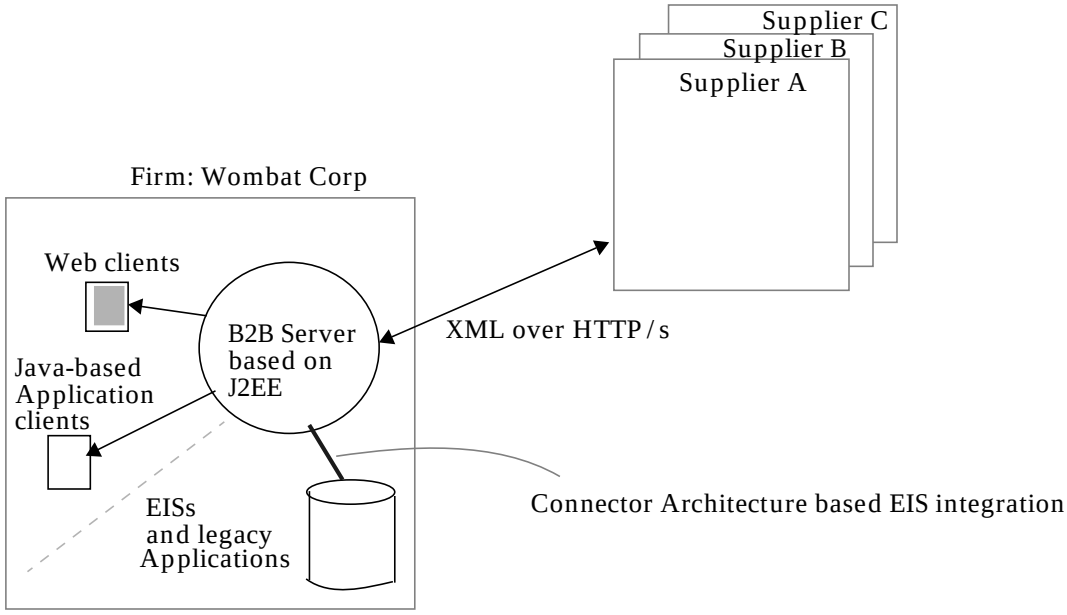
Wombat Corp. is a manufacturing firm that aims to adopt an e-business strategy. Wombat has huge existing investments in its EIS systems. The EISs includes ERP system and mainframe transaction processing systems.

Wombat needs to drive business-to-business interactions with its multiple supplier vendors. It wants to leverage its existing EIS investment while adopting the new e-business architecture.

Wombat buys a J2EE based server (called B2B server) from B2B, Inc. The B2B server supports ability to drive B2B interactions with multiple buyers / suppliers. The B2B interactions are driven using XML over HTTP / s.

The connector architecture enables Wombat to integrate its existing EISs with the B2B server. Wombat buys off-the-shelf resource adapters for its existing set of EISs. It then integrates its B2B server and applications (deployed on the B2B server) with its EISs using these resource adapters.

**FIGURE 4.0** Connector Architecture in B2B scenario




The applications deployed on the B2B server extract data from the underlying EISs. The extracted data may be directly in an XML format or can be converted by the applications to the XML format. The loosely-coupled B2B interactions with suppliers are then driven by exchanging XML data over HTTP/s protocol.

# 5 Connection Management

This chapter specifies the connection management contract between an application server and a resource adapter. It introduces the concepts and mechanisms relevant to this contract, and delineates the responsibilities of the roles of the resource adapter provider and application server vendor, in terms of their system-level support for the connection management contract. To complete the description of the connection management contract, this chapter also refers to the responsibilities of the application component provider and deployer. The chapter includes scenarios to illustrate the connection management contract.

## 5.1 Overview

An application component uses a connection factory to access a connection instance, which the component then uses to connect to the underlying EIS. A resource adapter acts as a factory of connections. Examples of connections include database connections, JMS (Java Message Service) connections, and SAP R/3 connections. Note that the support for pluggability of JMS providers into an application server will be added in the future versions of the specification.

Connection pooling manages connections that are expensive to create and destroy. Connection pooling of expensive connections leads to better scalability and performance in an operational environment. The connection management contract provides support for connection pooling.

## 5.2 Goals

The connection management contract has been designed with the following goals:

- To provide a consistent application programming model for connection acquisition for both managed and non-managed (two-tier) applications.
- To enable a resource adapter to provide a connection factory and connection interfaces based on the CCI specific to the type of resource adapter and EIS. This enables JDBC drivers to be aligned with the connector architecture with minimum impact on the existing JDBC APIs.
- To provide a generic mechanism by which an application server can provide different quality of services (QoS)—transactions, security, advanced pooling, error tracing/logging—for its configured set of resource adapters.
- To provide support for connection pooling.

The goal of the connector architecture is to enable efficient, scalable, and extensible connection pooling mechanisms, not to specify a mechanism or implementation for connection pooling. The goal is accomplished by defining a standard architected contract for connection management with the providers of connections—that is, resource adapters. An application server should use the connection management contract to implement a connection pooling mechanism in its own implementation-specific way.

## 5.3 Architecture: Connection Management

The connection management contract specifies an architected contract between an application server and a resource adapter. This connection management contract is shown with bold flow lines in the diagram Figure 5.0 on page 20. It includes the set of interfaces shown in the architecture diagram.

### Overview: Managed Application Scenario

The application server uses the deployment descriptor mechanism (specified in the section 10.6) to configure the resource adapter in the operational environment.

The resource adapter provides connection and connection factory interfaces. A connection factory acts as a factory for EIS connections. For example, `javax.sql.DataSource` and `java.sql.Connection` interfaces are JDBC-based interfaces for connectivity to a relational database.

The CCI (specified in chapter 9) defines `javax.resource.cci.ConnectionFactory` and `javax.resource.cci.Connection` as interfaces for a connection factory and a connection respectively.

The application component does a lookup of a connection factory in the JNDI name space. It uses the connection factory to get a connection to the underlying EIS. The connection factory instance delegates the connection creation request to the `ConnectionManager` instance.

The `ConnectionManager` enables the application server to provide different quality of services in the managed application scenario. These quality of services include transaction management, security, error logging and tracing, and connection pool management. The application server provides these services in its own implementation-specific way. The connector architecture does not specify how the application server implements these services.

The `ConnectionManager` instance, on receiving a connection creation request from the connection factory, does a lookup in the connection pool provided by the application server. If there is no connection in the pool that can satisfy the connection request, the application server uses the `ManagedConnectionFactory` interface (implemented by the resource adapter) to create a new physical connection to the underlying EIS. If the application server finds a matching connection in the pool, then it uses the matching `ManagedConnection` instance to satisfy the connection request.

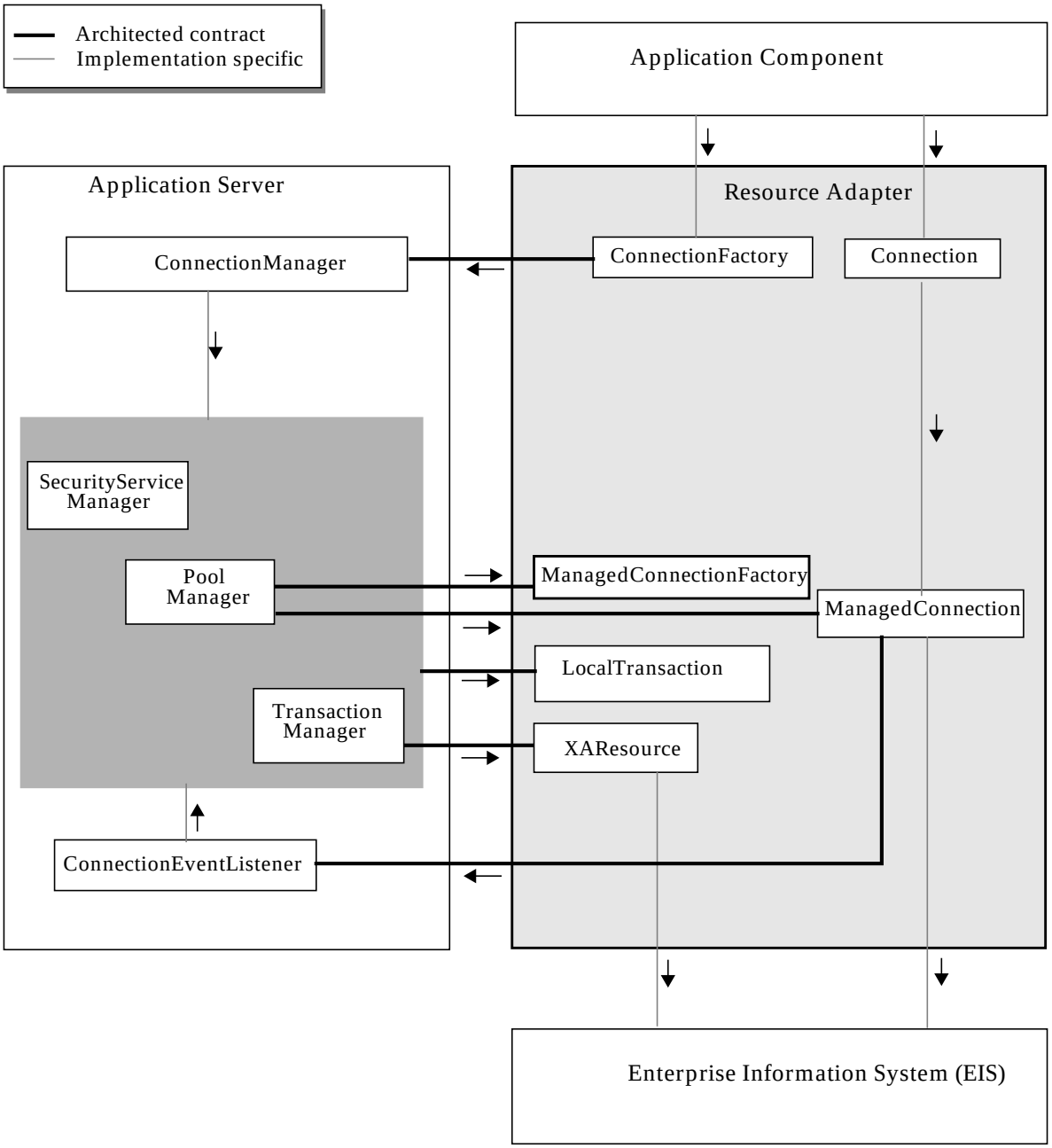
If a new `ManagedConnection` instance is created, the application server adds the new `ManagedConnection` instance to the connection pool.

The application server registers a `ConnectionEventListener` with the `ManagedConnection` instance. This listener enables application server to get event notifications related to the state of the `ManagedConnection` instance. The application server uses these notifications to manage connection pooling, manage transactions, cleanup connections, and handle any error conditions.

The application server uses the `ManagedConnection` instance to get a connection instance that acts as an application-level handle to the underlying physical connection. An instance of type `javax.resource.cci.Connection` is an example of such a connection handle. An application component uses the connection handle to access EIS resources.

The resource adapter implements the `XAResource` interface to provide support for transaction management. The resource adapter also implements the `LocalTransaction` interface so that the application server can manage transactions internal to a resource manager. The chapter on transaction management describes this transaction management contract between the application server (and its transaction manager) and the resource adapter (and its underlying resource manager).

**FIGURE 5.0** Architecture Diagram: Managed Application scenario



## 5.4 Application Programming Model

The application programming model for getting an EIS connection is similar across both managed (application server based) and non-managed scenarios. The following sections explain a typical application programming model scenario.

### 5.4.1 Managed Application scenario

The following steps are involved in a managed scenario:

- The application assembler or component provider specifies connection factory requirements for an application component using a deployment descriptor mechanism. For example, a bean provider specifies the following elements in the deployment descriptor for a connection factory reference. Note that the connection factory reference is part of the deployment descriptor for EJB components and not the resource adapter. Refer EJB specification [1] for details on the deployment mechanism for EJB components:
  - **`res-ref-name`**: `eis/MyEIS`
  - **`res-type`**: `javax.resource.cci.ConnectionFactory`
  - **`res-auth`**: `Application` or `Container`
- The deployer, using a resource adapter deployment tool, sets the configuration information (example: server name, port number) for the resource adapter. The application server uses a configured resource adapter to create physical connections to the underlying EIS. Refer chapter 10 for details on packaging and deployment of a resource adapter.
- The application component looks up a connection factory instance in the component's environment using the JNDI interface.

```
// obtain the initial JNDI Naming context
Context initctx = new InitialContext();

// perform JNDI lookup to obtain the connection factory
javax.resource.cci.ConnectionFactory cxf =
            (javax.resource.cci.ConnectionFactory)
                   initctx.lookup("java:comp/env/eis/MyEIS");
```

  The JNDI name passed in the method `NamingContext.lookup` is the same as that specified in `res-ref-name` element of the deployment descriptor. The JNDI lookup results in a connection factory instance of type `java.resource.cci.ConnectionFactory` as specified in the `res-type` element.
- The application component invokes the `getConnection` method on the connection factory to get an EIS connection. The returned connection instance represents an application-level handle to an underlying physical connection.

  An application component obtains multiple connections by calling the method `getConnection` on the connection factory multiple times.

```
javax.resource.cci.Connection cx = cxf.getConnection();
```

- The application component uses the returned connection to access the underlying EIS. The chapter on CCI specifies in detail the application programming model for EIS access.
- After the component finishes with the connection, it closes the connection using the `close` method on the `Connection` interface.

```
cx.close();
```

- If an application component fails to close an allocated connection after its use, that connection is considered an unused connection. The application server manages the cleanup of unused connections. When a container terminates a component instance, the container cleans up all connections used by that component instance. Refer section 5.5.4 and scenario 5.8.3 for details on the cleanup of connections.

### 5.4.2 Non-managed Application scenario

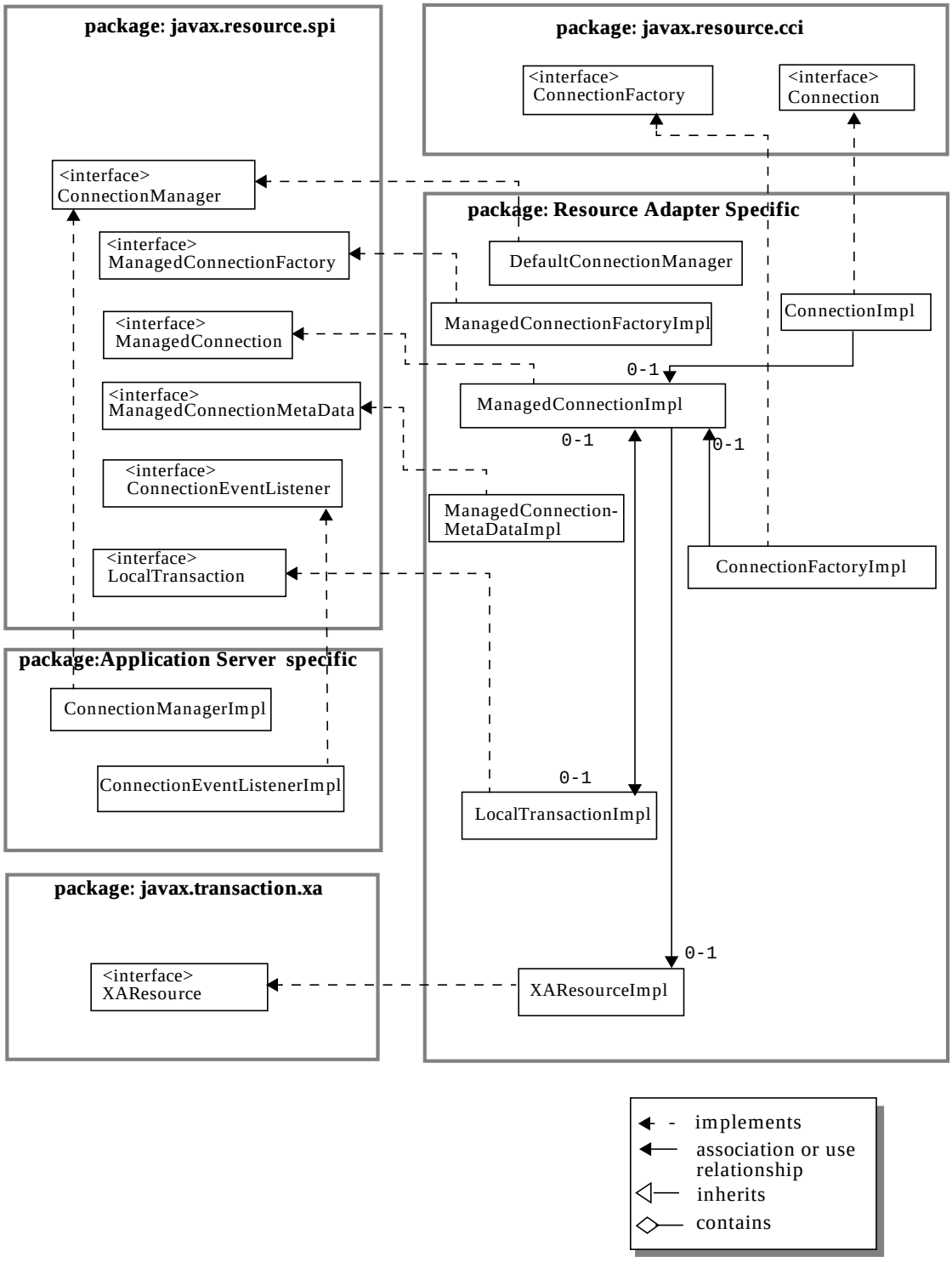
In a non-managed application scenario, the application developer follows a similar programming model to the managed application scenario. The non-managed case involves looking up of a connection factory instance, getting an EIS connection, using the connection for EIS access, and finally closing the connection.

## 5.5 Interface/Class specification

This section specifies the Java classes / interfaces defined as part of the connection management contract. For a complete specification of these classes / interfaces, refer to the Javadocs distributed with this document.

Figure 6.0 shows the class hierarchy for the connection management contract. The diagram also illustrates the responsibilities for the definition of an interface and its implementation:

**FIGURE 6.0** Class Diagram: Connection Management Architecture



### 5.5.1 ConnectionFactory and Connection[1]

A connection factory provides an interface to get a connection to an EIS instance. A connection provides connectivity to an underlying EIS.

One goal of the connector architecture is to support a consistent application programming model across both CCI and EIS specific client APIs. To achieve this goal, the connector architecture recommends a design pattern (specified as an interface template) for both connection factory and connection interfaces.

The CCI connection factory and connection interfaces (defined in the package `javax.resource.cci`) are based on the above design pattern. Refer 9.5 for details on the CCI connection factory and connection interfaces. The following code extract shows the CCI interfaces:

```
public interface javax.resource.cci.ConnectionFactory
     extends java.io.Serializable, javax.resource.Referenceable {

     public javax.resource.cci.Connection getConnection()
                  throws javax.resource.ResourceException;
     ...
}

public interface javax.resource.cci.Connection {
     public void close() throws javax.resource.ResourceException;
     ...
}
```

An example of a non-CCI interface is a resource adapter that uses the package `com.myeis` for its EIS specific interfaces, as follows:

```
public interface com.myeis.ConnectionFactory
     extends java.io.Serializable, javax.resource.Referenceable {

     public com.myeis.Connection getConnection()
                  throws com.myeis.ResourceException;
     ...
}

public interface com.myeis.Connection {
     public void close() throws com.myeis.ResourceException;
     ...
}
```

The JDBC interfaces—`javax.sql.DataSource`, `java.sql.Connection`—are examples of non-CCI connection factory and connection interfaces.

Note that the methods defined on a non-CCI interface are not required to throw a `ResourceException`. The exception can be specific to a resource adapter, for example: `java.sql.SQLException` for JDBC [3] interfaces.

1.In this document, the term `Physical Connection` refers to a `ManagedConnection` instance, while the term `Connection Handle` refers to an application-level connection handle. When the distinction between `Physical Connection` and `Connection Handle` is not important, the term `Connection` is used to refer to an EIS connection.

The following are additional guidelines for the recommended interface template:

- A resource adapter is allowed to add additional `getConnection` methods to its definition of a connection factory interface. These additional methods are specific to a resource adapter and its EIS. For example, CCI defines a variant of `getConnection` method that takes `java.resource.cci.ConnectionSpec` as a parameter.
- A resource adapter should only introduce additional `getConnection` methods if it requires additional flexibility (beyond that offered by the default `getConnection` method) in the connection request invocations.
- A connection interface is required to provide a method to close the connection. The behavior of such an application-level connection close is described in the OID Figure 11.0 on page 44.

The above design pattern leads to a consistent application programming model for connection creation and connection closing.

### Requirements

A resource adapter is required to provide implementations for both connection factory and connection interfaces.

> **Note**: In the connector architecture, a resource adapter provides an implementation of the connection factory interface in both managed and non-managed scenarios. This differs from the JDBC 2.0 [3] architecture.
>
> In the JDBC 2.0 architecture, an application server provides the implementation of `javax.sql.DataSource` interface. Using a similar design approach for the connector architecture will have required an application server to provide implementations of various connection factory interfaces defined by different resource adapters. Since connection factory interface may be defined as specific to an EIS, the application server may find difficult to provide implementations of connection factory interfaces without any code generation.

The connection factory implementation class delegates the `getConnection` method invocation from an application component to the associated `ConnectionManager` instance. The `ConnectionManager` instance is associated with a connection factory instance at its instantiation time [refer to the OID shown in Figure 39.0 on page 137].

Note that the connection factory implementation class is required to call the `ConnectionManager.allocateConnection` method in the same thread context in which the application component had called the `getConnection` method.

The connection factory implementation class is responsible for taking connection request information and passing it in a form required by the `ConnectionManager.allocateConnection` method.

```
public interface javax.resource.spi.ConnectionManager
            extends java.io.Serializable {

      public Object allocateConnection(
            ManagedConnectionFactory mcf,
            ConnectionRequestInfo cxRequestInfo)
                  throws ResourceException;
}

public interface javax.resource.spi.ConnectionRequestInfo {
      public boolean equals(Object other);
      public int hashCode();
}
```

### ConnectionRequestInfo

The `ConnectionRequestInfo` parameter to the `ConnectionManager.allocateConnection` method enables a resource adapter to pass its own request-specific data structure across the connection request flow.

A resource adapter extends the `ConnectionRequestInfo` interface to support its own data structure for the connection request.

A typical use allows a resource adapter to handle application component-specified per-connection request properties (for example, `client ID` and `language`). The application server passes these properties to `match/createManagedConnection` calls on the resource adapter. These properties remain opaque to the application server during the connection request flow.

It is important to note that the properties passed through the `ConnectionRequestInfo` instance should be client-specific (example: user name, password, language) and not related to the configuration of a target EIS instance (example: port number, server name).

The `ManagedConnectionFactory` instance is configured with properties required for the creation of a connection to a specific EIS instance. Note that a configured `ManagedConnectionFactory` instance must have the complete set of properties that are needed for the creation of the physical connections. This enables the container to manage connection request without requiring an application component to pass any explicit connection parameters. Configured properties on a `ManagedConnectionFactory` can be overridden through `ConnectionRequestInfo` in cases when component provides client-specific properties in the `getConnection` method invocation. Refer section 10.4.1 for details on the configuration of a `ManagedConnectionFactory`.

When the `ConnectionRequestInfo` reaches the `match/createManagedConnection` methods on the `ManagedConnectionFactory` instance, the resource adapter uses this additional per-request information to do connection creation and matching.

A resource adapter is required to implement the `equals` and `hashCode` methods defined on the `ConnectionRequestInfo` interface. The equality must be defined on the complete set of properties for the `ConnectionRequestInfo` instance. An application server can use these methods to structure its connection pool in an implementation specific way. Since `ConnectionRequestInfo` represents a resource adapter specific data structure, the conditions for equality are defined and implemented by a resource adapter.

### Additional Requirements

A resource adapter implementation is not required to support the mechanism for passing resource adapter-specific connection request information. It can choose to pass `null` for `ConnectionRequestInfo` in the `allocateConnection` invocation.

An implementation class for a connection factory interface is required to implement `java.io.Serializable`. This enables a connection factory instance to be stored in the JNDI naming environment. A connection factory implementation class is required to implement the interface `javax.resource.Referenceable`. Note that the `javax.resource.Referenceable` interface extends the `javax.naming.Referenceable` interface. Refer to section 10.5.3 for details on the JNDI reference mechanism.

A connection implementation class implements its methods in a resource adapter implementation-specific way. It must use `javax.resource.spi.ManagedConnection` instance as its underlying physical connection.

## 5.5.2 ConnectionManager

The `javax.resource.spi.ConnectionManager` provides a hook for a resource adapter to pass a connection request to an application server. An application server provides different quality of services as part of its handling of the connection request.

### Interface

The connection management contract defines a standard interface for the `ConnectionManager` as follows:

```
public interface javax.resource.spi.ConnectionManager
     extends java.io.Serializable {

     public Object allocateConnection(
            ManagedConnectionFactory mcf,
            ConnectionRequestInfo cxRequestInfo)
                   throws ResourceException;
}
```

The method `allocateConnection` is called by a resource adapter's connection factory instance so that the instance can delegate a connection request to the `ConnectionManager` instance.

The `ConnectionRequestInfo` parameter represents information specific to a resource adapter to handle the connection request.

### Requirements

An application server is required to provide implementation of the `ConnectionManager` interface. This implementation is not specific to any particular resource adapter or connection factory interface.

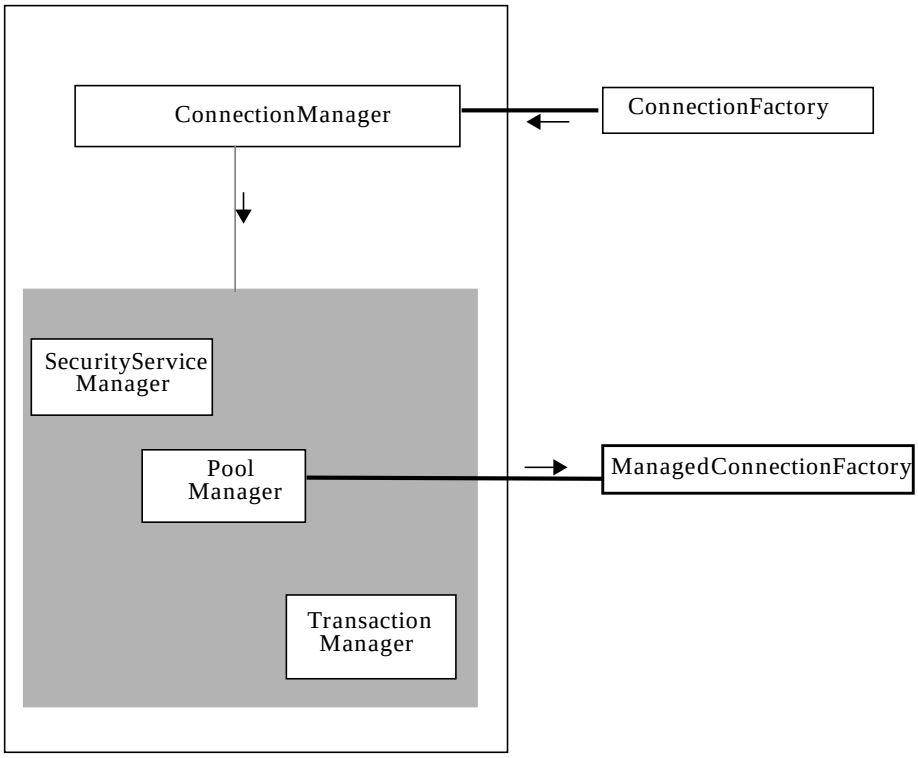
The `ConnectionManager` implementation delegates to internal mechanisms of an application server that provider quality of services (QoS)—security, connection pool management, transaction management, and error logging / tracing.

An application server should implement these services in a generic manner, independent of any resource adapter and EIS-specific mechanisms. The connector architecture does not specify how an application server implements these services; the implementation is specific to each application server.

After an application server *hooks-in* its services, the connection request is delegated to a `ManagedConnectionFactory` instance either for the creation of a new physical connection or for the matching of an already existing physical connection.

An implementation class for `ConnectionManager` interface is required to implement the `java.io.Serializable` interface.

**FIGURE 7.0** ConnectionManager and Application Server specific services




### 5.5.3 ManagedConnectionFactory

A `javax.resource.spi.ManagedConnectionFactory` instance is a factory of both `ManagedConnection` and connection factory instances. This interface supports connection pooling by defining methods for matching and creating connections.

#### Interface

The following code extract shows the interface specification for the `ManagedConnectionFactory`.

```
public interface javax.resource.spi.ManagedConnectionFactory
     extends java.io.Serializable {

     public Object createConnectionFactory(
           ConnectionManager connectionManager)
                                 throws ResourceException;

     public Object createConnectionFactory()
                                 throws ResourceException;

     public ManagedConnection createManagedConnection(
           javax.security.auth.Subject subject,
           ConnectionRequestInfo cxRequestInfo)
                                 throws ResourceException;

     public ManagedConnection matchManagedConnections(
           java.util.Set connectionSet,
```

```
            javax.security.auth.Subject subject,
            ConnectionRequestInfo cxRequestInfo)
                   throws ResourceException;

      public boolean equals(Object other);
      public int hashCode();

}
```

The method `createConnectionFactory` creates a connection factory instance. For CCI, the connection factory instance is of the type `javax.resource.cci.ConnectionFactory`. The connection factory instance is initialized with the `ConnectionManager` instance provided by the application server.

When the `createConnectionFactory` method takes no arguments, `ManagedConnectionFactory` provides a default `ConnectionManager` instance. This case is used in a non-managed application scenario.

The method `createManagedConnection` creates a new physical connection to the underlying EIS instance. The `ManagedConnectionFactory` uses the security information (passed as a `Subject` instance) and an optional `ConnectionRequestInfo` to create this new physical connection [refer to security contract in chapter 8 for more details].

A created `ManagedConnection` instance typically maintains internal information about the security context (under which the connection has been created) and any connection-specific parameters (for example, socket connection).

The method `matchManagedConnections` enables application server to use a resource adapter-specific criteria for matching a `ManagedConnection` instance to service a connection request. Application server finds a candidate set of `ManagedConnection` instances from its connection pool based on application server-specific criteria and passes this candidate set to the `matchManagedConnections` method.

The method `matchManagedConnections` matches a candidate set of connections using criteria known internally to the resource adapter. The criteria used for matching is specific to a resource adapter and is not specified by the connector architecture.

A `ManagedConnection` instance has specific internal state, in terms of its security context and physical connection-specific state. The `ManagedConnectionFactory` implementation compares this information for each `ManagedConnection` instance in the candidate set against the information passed in through the `matchManagedConnections` method and the configuration of *this* `ManagedConnectionFactory` instance. The `ManagedConnectionFactory` uses the results of this comparison to choose the `ManagedConnection` instance that can best satisfy the current connection request.

If the resource adapter cannot find an acceptable `ManagedConnection` instance, it returns a `null`. In this case, the application server requests the resource adapter to create a new connection instance.

### Requirements

A resource adapter is required to provide an implementation of the `ManagedConnectionFactory` interface.

It is required that the `ManagedConnectionFactory` implementation class extend the implementation of the `hashCode` and `equals` methods defined in the `java.lang.Object` class. These two methods are used by an application server to structure its connection pool in an implementation-specific way. The `equals` and `hashCode` method implementation should be based on a complete set of configuration properties that makes a `ManagedConnectionFactory` instance unique and specific to an EIS instance.

An implementation class for `ManagedConnectionFactory` interface is required to implement the `java.io.Serializable` interface.

### Connection Pool Implementation

The connector architecture does not specify how an application server implements connection pooling. However, it recommends that an application server should structure its connection pool such that it uses the connection creation / matching facility in an efficient manner and does not cause resource starvation.

The following paragraphs provide non-prescriptive guidelines for connection pool implementation by an application server.

An application server may partition its pool on a per `ManagedConnectionFactory` instance (and thereby on a per EIS instance) basis. An application server may choose to guarantee (in an implementation specific way) that it will always partition connection pools with at least per `ManagedConnectionFactory` instance granularity.

The per-`ManagedConnectionFactory` instance pool may be further partitioned based on the transaction or security context or any client-specific parameters (as associated with the `ConnectionRequestInfo`). When an application server calls the matching facility, it is recommended that the application server narrows down the candidate set of `ManagedConnection` instances to a reasonable limit and achieves matching efficiently. For example, an application server may pass only those `ManagedConnection` instances to the `matchManagedConnection` method that are associated with the target `ManagedConnectionFactory` instance (and thereby a specific target EIS instance).

An application server may use additional parameters for its search and matching criteria used in its connection pool management. These parameters may be EIS or application server specific. The `equals` and `hashCode` methods defined on both `ManagedConnectionFactory` and `ConnectionRequestInfo` facilitate the connection pool management and structuring by an application server.

### Requirement for XA Recovery

The `ManagedConnectionFactory` implementation for a XA-capable resource adapter (refer chapter 6 for more details on transactions) is required to support `createManagedConnection` method that takes a `Subject` and a `null` for the parameter `ConnectionRequestInfo`. This enables the application server to get a `XAResource` instance using `ManagedConnection.getXAResource` and then call `XAResource.recover` method. Note that the application server uses this special case only to get to the `XAResource` instance for the underlying resource manager.

The reason for this requirement is that application server may not have a valid `ConnectionRequestInfo` when it needs to get the `ManagedConnection` instance to initiate recovery. Refer section 8.2.6 for additional details on the `ManagedConnectionFactory.createManagedConnection` method.

## 5.5.4 ManagedConnection

A `javax.resource.spi.ManagedConnection` instance represents a physical connection to an underlying EIS.

> **Note**: The connector architecture allows one or more `ManagedConnection` instances to be multiplexed over a single *physical pipe* to an EIS. However, for simplicity, this specification describes a `ManagedConnection` instance as being mapped 1-1 to a physical connection.

The creation of a `ManagedConnection` instance typically results in the allocation of EIS and resource adapter resources (for example: memory, network socket) for each physical connection. Since these resources can be costly and scarce, an application server pools `ManagedConnection` instances in a managed environment.

Connection pooling improves the scalability of an application environment. An application server uses the ManagedConnectionFactory and ManagedConnection interfaces to implement connection pool management.

An application server also uses the transaction management-related methods (getXAResource and getLocalTransaction) on the ManagedConnection interface to manage transactions. These methods are discussed in more detail in the Transaction Management chapter.

The ManagedConnection interface also provides methods to support error logging and tracing in a managed environment.

### Interface

The connection management contract defines the following interface for a ManagedConnection. The following code extract shows only the methods that are used for connection pool management. The remaining methods are introduced in other parts of the specification.

```
public interface javax.resource.spi.ManagedConnection {
     public Object getConnection(
                  javax.security.auth.Subject subject,
                  ConnectionRequestInfo cxRequestInfo)
                                    throws ResourceException;
     public void destroy() throws ResourceException;
     public void cleanup() throws ResourceException;

     // Methods for Connection and transaction event notifications
     public void addConnectionEventListener(
                  ConnectionEventListener listener);
     public void removeConnectionEventListener(
                  ConnectionEventListener listener);

     public ManagedConnectionMetaData getMetaData()
                  throws ResourceException;

     // Additional methods - specified in the other sections
     ...
}
```

The getConnection method creates a new application-level connection handle. A connection handle is tied to an underlying physical connection represented by a ManagedConnection instance. For CCI, the connection handle created by a ManagedConnection instance is of the type javax.resource.cci.Connection. A connection handle is tied to its ManagedConnection instance in a resource adapter implementation-specific way.

A ManagedConnection instance may use the getConnection method to change the state of the physical connection based on the Subject and ConnectionRequestInfo arguments. For example, a resource adapter can re-authenticate a physical connection to the underlying EIS when the application server calls the getConnection method. Section 8.2.7 specifies re-authentication requirements in more detail.

The method addConnectionEventListener allows a connection event listener to register with a ManagedConnection instance. The ManagedConnection instance notifies connection close / error and local transaction-related events to its registered set of listeners.

The removeConnectionEventListener method removes a registered ConnectionEventListener instance from a ManagedConnection instance.

The method getMetaData returns the metadata information (represented by the ManagedConnectionMetaData interface) for a ManagedConnection and the connected EIS instance.

### Connection Sharing and Multiple Connection Handles

To support connection sharing, the application server can call `getConnection` multiple times on a `ManagedConnection` instance. In this case, a call to the method `ManagedConnection.getConnection` does not invalidate any previously created connection handles. Multiple connection handles can exist concurrently for a single `ManagedConnection` instance. This design supports the connection sharing mechanism. Note that the application server may choose to provide connection sharing, but is not required to do so. Refer to section 6.9 for more details.

Because multiple connection handles to a single `ManagedConnection` can exist concurrently, a resource adapter implementation may:

- Ensure that there is at most one connection handle associated actively with a `ManagedConnection` instance. The active connection handle is the only connection using the `ManagedConnection` instance until an application-level `close` is called on this connection handle. For example, a `ManagedConnection.getConnection` method implementation associates a newly created connection handle as the active connection handle. Any operations on the `ManagedConnection` from any previously created connection handles should result in an application level exception. An example application level exception extends the `javax.resource.ResourceException` interface and is specific to a resource adapter. A scenario illustrating this implementation is shown in the Scenario: Local Transaction on page 73.
- Provide thread-safe semantics for a `ManagedConnection` implementation to support concurrent access to a `ManagedConnection` instance from multiple connection handles.

### Connection Matching Contract

The application server invokes `ManagedConnectionFactory.matchManagedConnection` method (implemented by a resource adapter) to find a matching `ManagedConnection` for servicing a connection request. Application server passes a candidate set of `ManagedConnection` instances to the `matchManagedConnection` method.

The application server should use connection matching contract for `ManagedConnection` instances that have no existing connection handles. A candidate set passed to `matchManagedConnection` method should not have any `ManagedConnection` instance with existing connection handles.

There is no requirement that `matchManagedConnections` implementation be capable of performing a match across a candidate set that includes `ManagedConnection` instances with existing connection handles. Note that a resource adapter can return a successful match with the requirement that `ManagedConnection.getConnection` method will later change the state of the matched `ManagedConnection`. To avoid any unexpected matching behavior, the application server should not pass a `ManagedConnection` instance with existing connection handles to the `matchManagedConnections` method as part of a candidate set.

A connection request can lead to creation of additional connection handle for a `ManagedConnection` instance that already has one or more existing connection handles. In this case, the application server should take the responsibility of checking whether or not the chosen `ManagedConnection` instance can service such a request. Refer section Connection Sharing on page 71 for details.

### Cleanup of ManagedConnection

A resource adapter typically allocates system resources (outside a JVM) for a `ManagedConnection` instance. Additionally, a `ManagedConnection` instance can have state specific to a client, such as security context and data / function access structures (query result set is an example).

The method `ManagedConnection.cleanup` initiates a cleanup of any client-specific state maintained by a `ManagedConnection` instance. The `cleanup` is required to invalidate all connection handles created using this `ManagedConnection` instance. Any attempt by an applica-

tion component to use associated connection handle after cleanup of the underlying ManagedConnection should result in an exception.

The container always drives the cleanup of a ManagedConnection instance. The container keeps track of created connection handles in an implementation specific mechanism. It invokes ManagedConnection.cleanup when it has to invalidate all connection handles (associated with this ManagedConnection instance) and put the ManagedConnection instance back in to the pool. This may be called after the end of a connection sharing scope or when the last associated connection handle is closed for a ManagedConnection instance.

The invocation of the ManagedConnection.cleanup method on an already cleaned-up connection should not throw an exception.

The cleanup of a ManagedConnection instance resets its client-specific state and prepares the connection to be put back into a connection pool. The cleanup method should not cause the resource adapter to close the physical *pipe* and reclaim system resources associated with the physical connection.

An application server should explicitly call ManagedConnection.destroy to destroy a physical connection. An application server should destroy a physical connection to manage the size of its connection pool and to reclaim system resources.

A resource adapter should destroy all allocated system resources for this ManagedConnection instance when the method destroy is called.

#### Requirements

A resource adapter is required to provide an implementation of the ManagedConnection interface.

### 5.5.5 ManagedConnectionMetaData

The method ManagedConnection.getMetaData returns a javax.resource.spi.ManagedConnectionMetaData instance. The ManagedConnectionMetaData provides information about a ManagedConnection and the connected EIS instance. This information is only available to the caller of this method if a valid physical connection exists for an EIS instance.

#### Interface

The ManagedConnectionMetaData interface provides the following information about an EIS instance:

- Product name of the EIS instance
- Product version of the EIS instance
- Maximum number of concurrent connections from different processes that an EIS instance can support
- User name for this connection, as know to the EIS instance

The method getUserName returns the user name known to the underlying EIS instance for an active connection. The name corresponds to the resource principal under whose security context the connection to the EIS instance has been established.

#### Requirements

A resource adapter is required to provide an implementation of the ManagedConnectionMetaData interface. An instance of this implementation class should be returned from the ManagedConnection.getMetaData method.

### 5.5.6 ConnectionEventListener

The connector architecture provides an event callback mechanism that enables an application server to receive notifications from a ManagedConnection instance. An application server uses these event notifications to manage its connection pool, to clean up invalid or

terminated connections, and to manage local transactions. The transaction management chapter discusses local transaction-related event notifications in more detail.

An application server implements the `javax.resource.spi.ConnectionEventListener` interface. It uses the `ManagedConnection.addConnectionEventListener` method to register a connection listener with a `ManagedConnection` instance.

**Interface**

The following code extract specifies the `ConnectionEventListener` interface:

```
public interface javax.resource.spi.ConnectionEventListener {
     public void connectionClosed(ConnectionEvent event);
     public void connectionErrorOccurred(ConnectionEvent event);

     // Local Transaction Management related events
     public void localTransactionStarted(ConnectionEvent event);
     public void localTransactionCommitted(ConnectionEvent event);
     public void localTransactionRolledback(ConnectionEvent event);
}
```

A `ManagedConnection` instance calls the `ConnectionEventListener.connectionClosed` method to notify its registered set of listeners when an application component closes a connection handle. The application server uses this connection close event to make a decision on whether or not to put the `ManagedConnection` instance back into the connection pool.

The `ManagedConnection` instance calls the `ConnectionEventListener.connectionErrorOccurred` method to notify its registered listeners of the occurrence of a physical connection-related error. The event notification happens just before a resource adapter throws an exception to the application component using the connection handle.

The `connectionErrorOccurred` method indicates that the associated `ManagedConnection` instance is now invalid and unusable. The application server handles the connection error event notification by initiating application server-specific cleanup (for example, removing `ManagedConnection` instance from the connection pool) and then calling `ManagedConnection.destroy` method to destroy the physical connection.

A `ManagedConnection` instance also notifies its registered listeners for transaction-related events by calling the following methods—`localTransactionStarted`, `localTransactionCommitted`, and `localTransactionRolledback`. An application server uses these notifications to manage local transactions. See section 6.7 for details on the local transaction management.

### 5.5.7 ConnectionEvent

A `javax.resource.spi.ConnectionEvent` class provides information about the source of a connection-related event. A `ConnectionEvent` instance contains the following information:

- Type of the connection event
- `ManagedConnection` instance that has generated the connection event. A `ManagedConnection` instance is returned from the `ConnectionEvent.getSource` method.
- Connection handle associated with the `ManagedConnection` instance; required for the `CONNECTION_CLOSED` event and optional for the other event types.
- Optionally, an exception indicating a connection related error. Refer 12.2 for details on the system exception. Note that exception is used for CONNECTION_ERROR_OCCURRED.

This class defines the following types of event notifications:

- CONNECTION_CLOSED

- LOCAL_TRANSACTION_STARTED
- LOCAL_TRANSACTION_COMMITTED
- LOCAL_TRANSACTION_ROLLEDBACK
- CONNECTION_ERROR_OCCURRED

## 5.6 Error Logging and Tracing

The connector architecture provides basic support for error logging and tracing in both managed and non-managed environments. This support enables an application server to detect errors related to a resource adapter and its EIS and to use error information for debugging.

### ManagedConnectionFactory

The `javax.resource.spi.ManagedConnectionFactory` interface defines the following methods for error logging and tracing:

```
public interface javax.resource.spi.ManagedConnectionFactory
     extends java.io.Serializable {

     public void setLogWriter(java.io.PrintWriter out)
                                     throws ResourceException;
     public java.io.PrintWriter getLogWriter()
                                     throws ResourceException;
     ...
}
```

The log writer is a character output stream to which all logging and tracing messages for a `ManagedConnectionFactory` instance are printed.

A character output stream can be registered with a `ManagedConnectionFactory` instance using the `setLogWriter` method. A `ManagedConnectionFactory` implementation uses this character output stream to output error log and trace information.

An application server manages the association of a log writer with a `ManagedConnectionFactory`. When a `ManagedConnectionFactory` instance is created, the log writer is initially `null` and logging is disabled. Associating a log writer with a `ManagedConnectionFactory` instance enables logging and tracing for the `ManagedConnectionFactory` instance.

An application server administrator primarily uses the error and trace information printed on a log writer by a `ManagedConnectionFactory` instance. This information is typically system-level in nature (example: information related to connection pooling and transactions) rather than of direct interest to application developers.

### ManagedConnection

The `javax.resource.spi.ManagedConnection` interface defines the following methods to support error logging and tracing specific to a physical connection.

```
public interface javax.resource.spi.ManagedConnection {
     public void setLogWriter(java.io.PrintWriter out)
                                     throws ResourceException;
     public java.io.PrintWriter getLogWriter()
                                     throws ResourceException;
     ...
}
```

A newly created `ManagedConnection` instance gets the default log writer from the `ManagedConnectionFactory` instance that creates the `ManagedConnection` instance. The default log

writer can be overridden by an application server using the `ManagedConnection.setLogWriter` method. The setting of the log writer on a `ManagedConnection` enables an application server to manage error logging and tracing specific to the physical connection represented by a `ManagedConnection` instance.

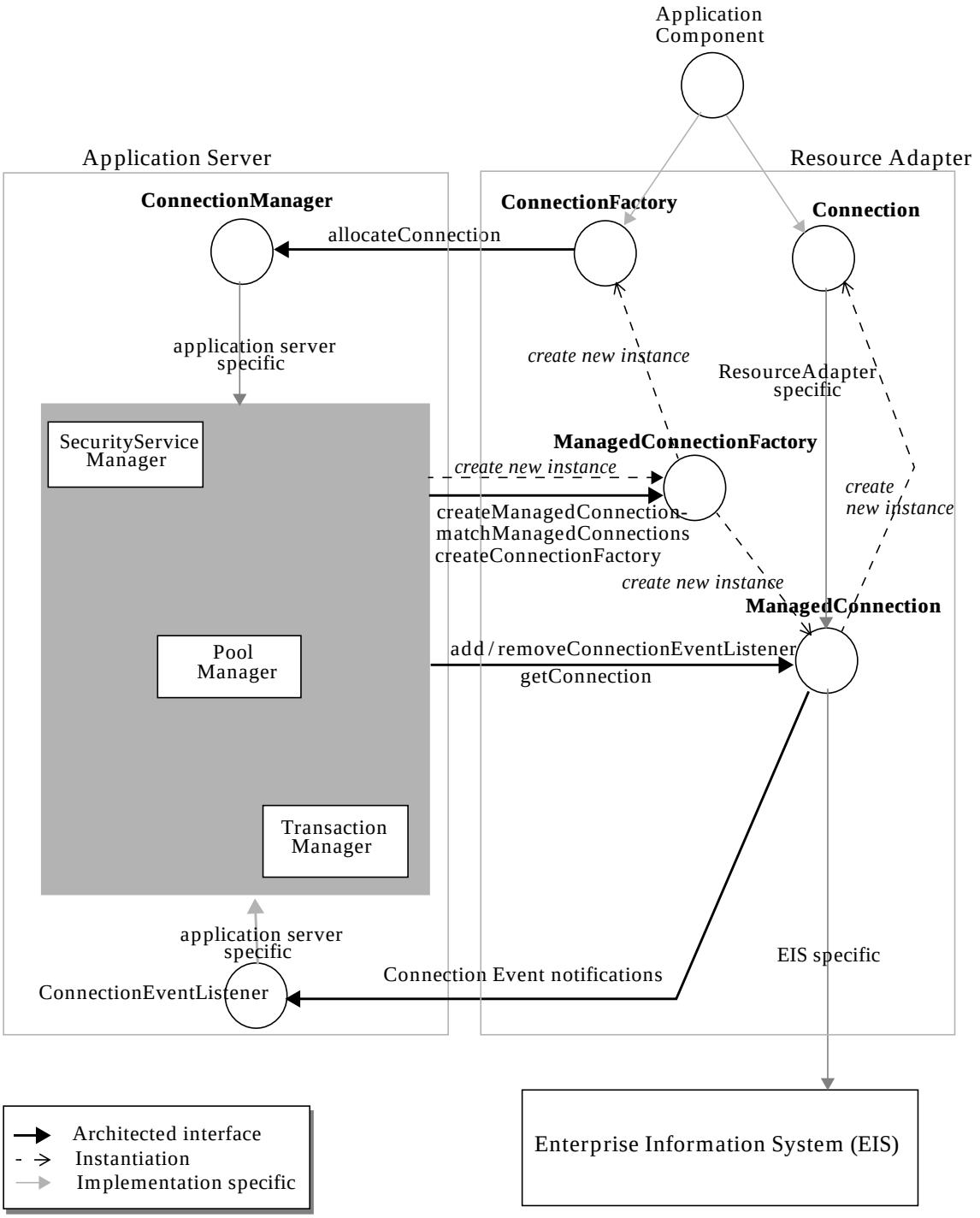
An application server can optionally choose to disassociate the log writer from a `ManagedConnection` instance (by using `setLogWriter` passing `null`) when this connection instance is put back into the connection pool.

## 5.7 Object Diagram

Figure 8.0 shows the object diagram for the connection management architecture. It shows invocations across the various object instances that correspond to the architected interfaces in the connection management contract, as opposed to those instances specific to implementations of the application server and the resource adapter.

To keep the diagram simple, it does not show the transaction management contract-related interfaces (`XAResource` and `LocalTransaction`) and invocations.

**FIGURE 8.0** Object Diagram: Connection Management architecture



## 5.8 Illustrative Scenarios

The following section uses sequence diagrams to illustrate various interactions between the object instances involved in the connection management contract.

Some sequence diagrams include a box labeled "Application Server". This box refers to various modules and classes internal to an application server. These modules and classes communicate through contracts that are application server implementation specific**.**

In this section, the CCI interfaces—`javax.resource.cci.ConnectionFactory` and `javax.resource.cci.Connection`—represent connection factory and connection interfaces respectively.

The description of these sequence diagrams does not include transaction-related details. These are covered in the Transaction Management chapter.

### 5.8.1 Scenario: Connection Pool Management

The following object interactions are involved in the scenario shown in Figure 9.0 on page 40:

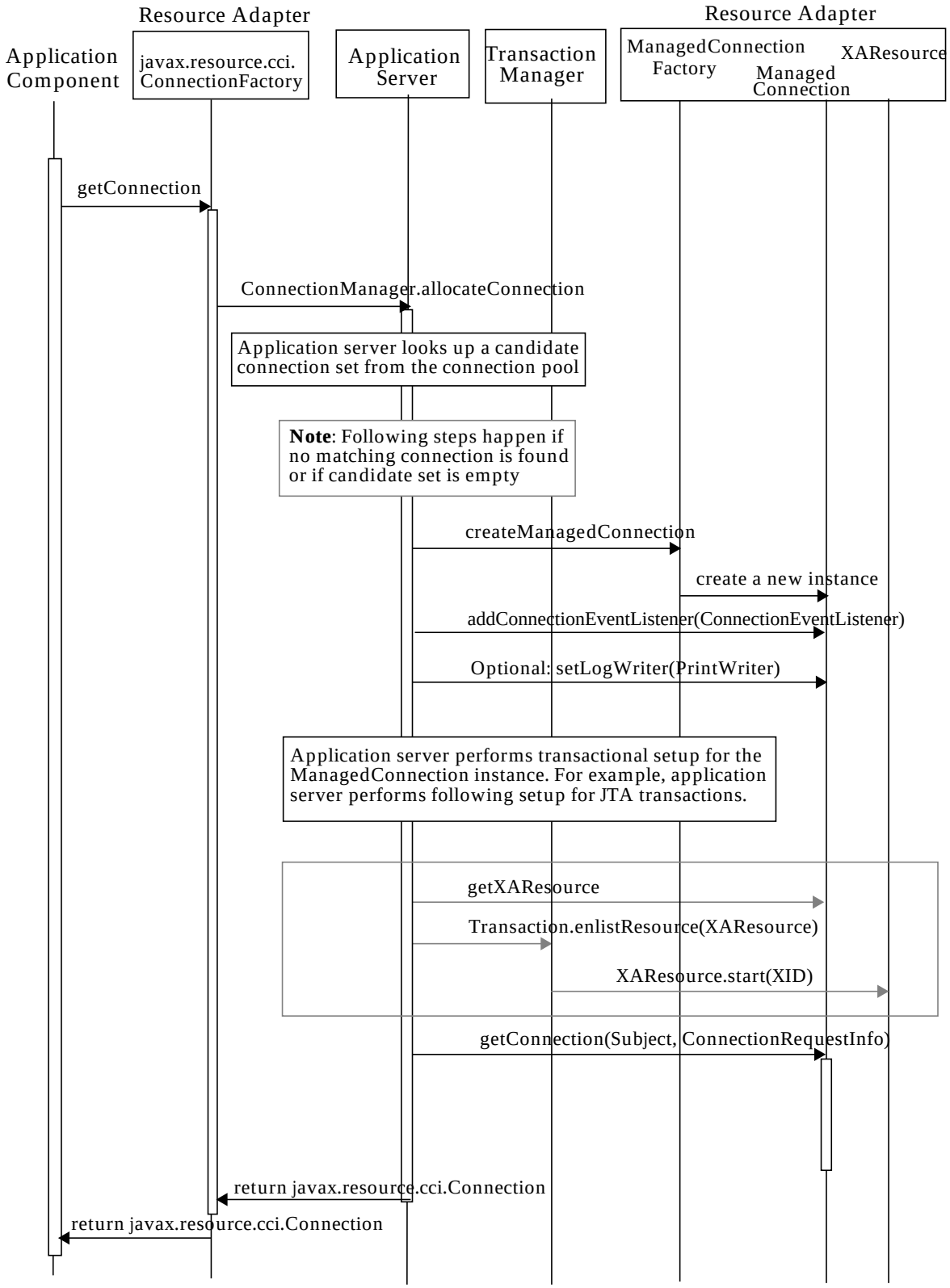
- The application component calls the `getConnection` method on the `javax.resource.cci.ConnectionFactory` instance (returned from the JNDI lookup) to get a connection to the underlying EIS instance. Refer to section 10.5 for details on the JNDI configuration and lookup.
- The `ConnectionFactory` instance initially handles the connection request from the application component in a resource adapter-specific way. It then delegates the connection request to the associated `ConnectionManager` instance. The `ConnectionManager` instance has been associated with the `ConnectionFactory` instance when the `ConnectionFactory` was instantiated.

  The `ConnectionFactory` instance receives all connection request information passed through the `getConnection` method and, in turn, passes it in a form required by the method `ConnectionManager.allocateConnection`. The `ConnectionRequestInfo` parameter to the `allocateConnection` method enables a `ConnectionFactory` implementation class to pass client-specific connection request information. This information is opaque to an application server and is used subsequently by a resource adapter to do connection matching and creation.
- The `ConnectionManager` instance (provided by the application server) handles the `allocateConnection` request by interacting with the application server-specific connection pool manager. The interaction between a `ConnectionManager` instance and pool manager is internal and specific to an application server.
- The application server finds a candidate set of `ManagedConnection` instances from its connection pool. The candidate set includes all `ManagedConnection` instances that the application server considers suitable for handling the current connection allocation request. The application server finds the candidate set using its own implementation-specific structuring and lookup criteria for the connection pool. Refer section 5.5.3 for guidelines of connection pool implementation by an application.
- If the application server finds no matching `ManagedConnection` instance that can best handle this connection allocation request, or if the candidate set is empty, the application server calls the `ManagedConnectionFactory.createManagedConnection` method to create a new physical connection to the underlying EIS instance. The application server passes necessary security information (as JAAS `Subject`) as part of this method invocation. For details on the security contract, refer to the `Security Management` chapter. It can also pass the `ConnectionRequestInfo` information to the resource adapter. The connection request information has been associated with the connection allocation request by the resource adapter and is used during connection creation.

- The `ManagedConnectionFactory` instance creates a new physical connection to the underlying EIS to handle the `createManagedConnection` method. This new physical connection is represented by a `ManagedConnection` instance. The `ManagedConnectionFactory` uses the security information (passed as a `Subject` instance), `ConnectionRequestInfo`, and its default set of configured properties (port number, server name) to create a new `ManagedConnection` instance. Refer to the security contract for more details on the `createManagedConnection` method.
- The `ManagedConnectionFactory` instance initializes the created `ManagedConnection` instance and returns it to the application server.
- The application server registers a `ConnectionEventListener` instance with the `ManagedConnection` instance, enabling it to receive notifications for events on this connection. The application server uses these event notifications to manage connection pooling and transactions.
- The `ManagedConnection` instance obtains its log writer (for error logging and tracing support) from the `ManagedConnectionFactory` instance that created this connection. However, an application server can set a new log writer with a `ManagedConnection` instance to do additional error logging and tracing at the level of a `ManagedConnection`.
- The application server does the necessary transactional setup for the `ManagedConnection` instance. The chapter on Transaction Management explains this step in more detail.
- Next, the application server calls `ManagedConnection.getConnection` method to get an application level connection handle of type `javax.resource.cci.Connection`. A `ManagedConnection` instance uses the `Subject` and `ConnectionRequestInfo` parameters to the `getConnection` method to change the state of the `ManagedConnection`.

  Calling the `getConnection` method does not necessarily create a new physical connection to the EIS instance. Calling `getConnection` produces a temporary connection handle that is used by an application component to access the underlying physical connection. The actual underlying physical connection is represented by a `ManagedConnection` instance.
- The application server returns the connection handle to the resource adapter. The resource adapter then passes the connection handle to the application component that initiated the connection request.

**FIGURE 9.0** OID: Connection Pool Management with new Connection Creation

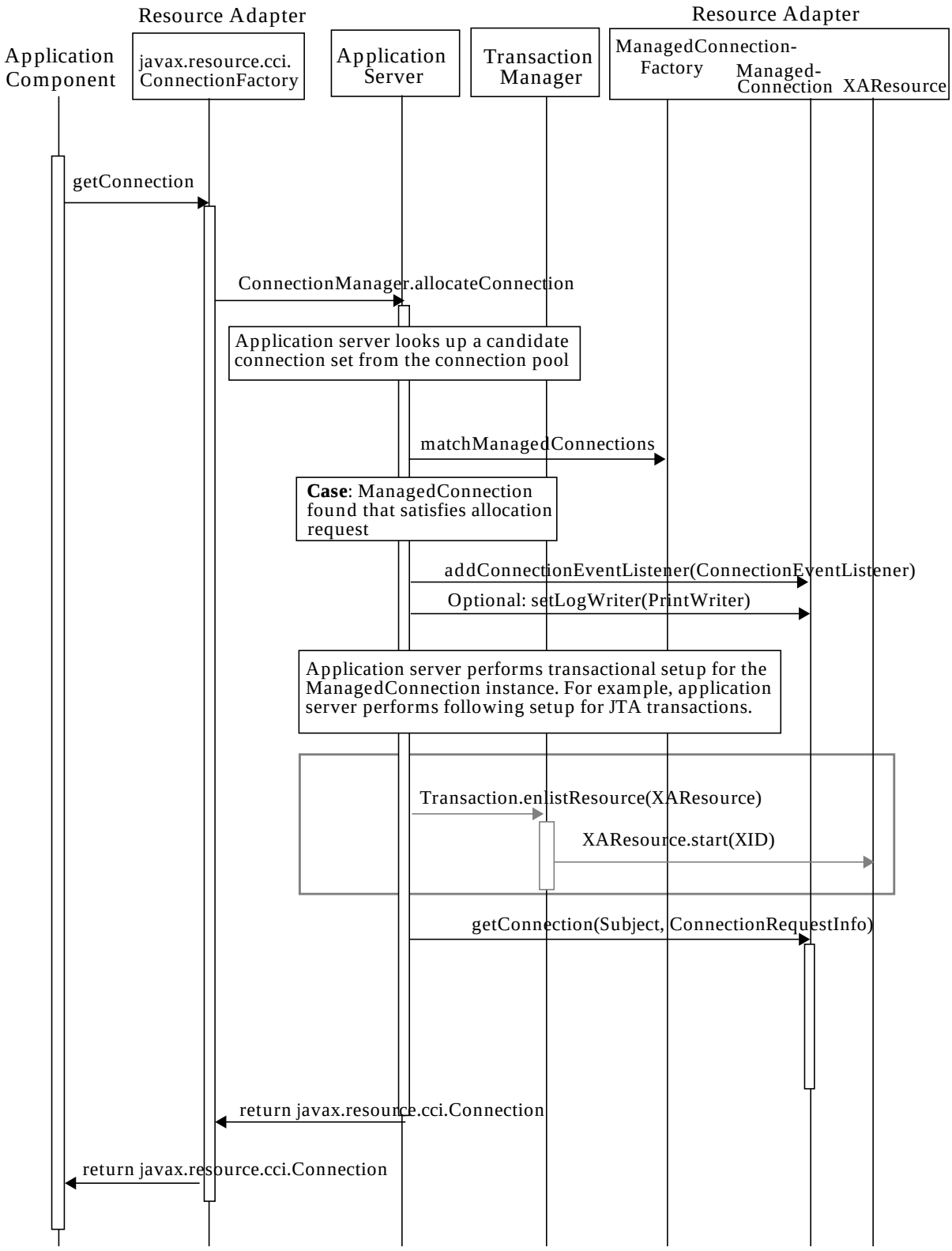


### 5.8.2 Scenario: Connection Matching

The OID on the page 42 shows the object interactions for a connection matching scenario—that is, a scenario in which the application server finds a non-empty candidate connection set and calls the resource adapter to do matching on the candidate set. The following steps are involved in this scenario:

- The application server handles the connection allocation request by creating a candidate set of `ManagedConnection` instances from the connection pool. The candidate set includes the `ManagedConnection` instances that the application server considers suitable for handling the current connection allocation request. The application server finds this candidate set using its own implementation-specific structuring and lookup criteria for the connection pool. Refer section 5.5.3 for guidelines of connection pool implementation by an application.
- The application server calls the `ManagedConnectionFactory.matchManaged-Connections` method to enable the resource adapter to do the connection matching. It passes the candidate connection set, security information (as a `Subject` instance associated with the current connection request), and any `ConnectionRequestInfo`.
- The `ManagedConnectionFactory` instance matches the candidate set of connections using the criteria known internally to the resource adapter. The `matchManagedConnections` method returns a `ManagedConnection` instance that the resource adapter considers to be an acceptable match for the current connection allocation request.
- The application server can set a new log writer with the `ManagedConnection` instance to do error logging and tracing at the level of the `ManagedConnection`.
- The application server does the necessary transactional setup for the `ManagedConnection` instance. The chapter on `Transaction Management` explains this step in more detail.
- The application server calls the `ManagedConnection.getConnection` method to get a new application level connection handle.
- The `ManagedConnection.getConnection` method implementation uses the `Subject` parameter and any `ConnectionRequestInfo` to set the state of the `ManagedConnection` instance based on the current connection allocation request. Refer to section 8.2.7 for details if a resource adapter implements support for re-authentication of a `ManagedConnection` instance.
- The application server returns the connection handle to the resource adapter. The resource adapter then passes the connection handle to the application component that initiated the connection request.

**FIGURE 10.0** OID: Connection Pool Management with Connection Matching



### 5.8.3 Scenario: Connection Event Notifications and Connection Close

For each `ManagedConnection` instance in the pool, the application server registers a `ConnectionEventListener` instance to receive close and error events on the connection. This scenario explains how the connection event callback mechanism enables an application server to manage connection pooling.

The scenario involves the following steps (see Figure 11.0 on page 44) when an application component initiates a connection close:

- The application component releases an allocated connection handle using the `close` method on the `javax.resource.cci.Connection` instance. The `Connection` instance delegates the close method to the associated `ManagedConnection` instance. The delegation happens through an association between `ManagedConnection` instance and the corresponding connection handle `Connection` instance. The mechanism by which this association is achieved is specific to the implementation of a resource adapter.
- The connection management contract places a requirement that a `ManagedConnection` instance must not alter the state of a physical connection while handling the connection close.
- The `ManagedConnection` instance notifies all its registered listeners of the application's connection close request using the `ConnectionEventListener.connectionClosed` method. It passes a `ConnectionEvent` instance with the event type set to CONNECTION_CLOSED.
- On receiving the connection close event notification, the application server performs the transaction management-related cleanup of the `ManagedConnection` instance. Refer to Figure 11.0 on page 44 for details on the cleanup of a `ManagedConnection` instance participating in a JTA transaction.
- The application server also uses the connection close event notification to manage its connection pool. On receiving the connection close notification, the application server calls the `ManagedConnection.cleanup` method to perform cleanup on the `ManagedConnection` instance that raised the connection close event. The application server-initiated cleanup of a `ManagedConnection` instance prepares this `ManagedConnection` instance to be reused for subsequent connection requests.
- After initiating the necessary cleanup for the `ManagedConnection` instance, the application server puts the `ManagedConnection` instance back into the connection pool. The application server should be able to use this available `ManagedConnection` instance to handle future connection allocation requests from application components.

#### Connection Cleanup

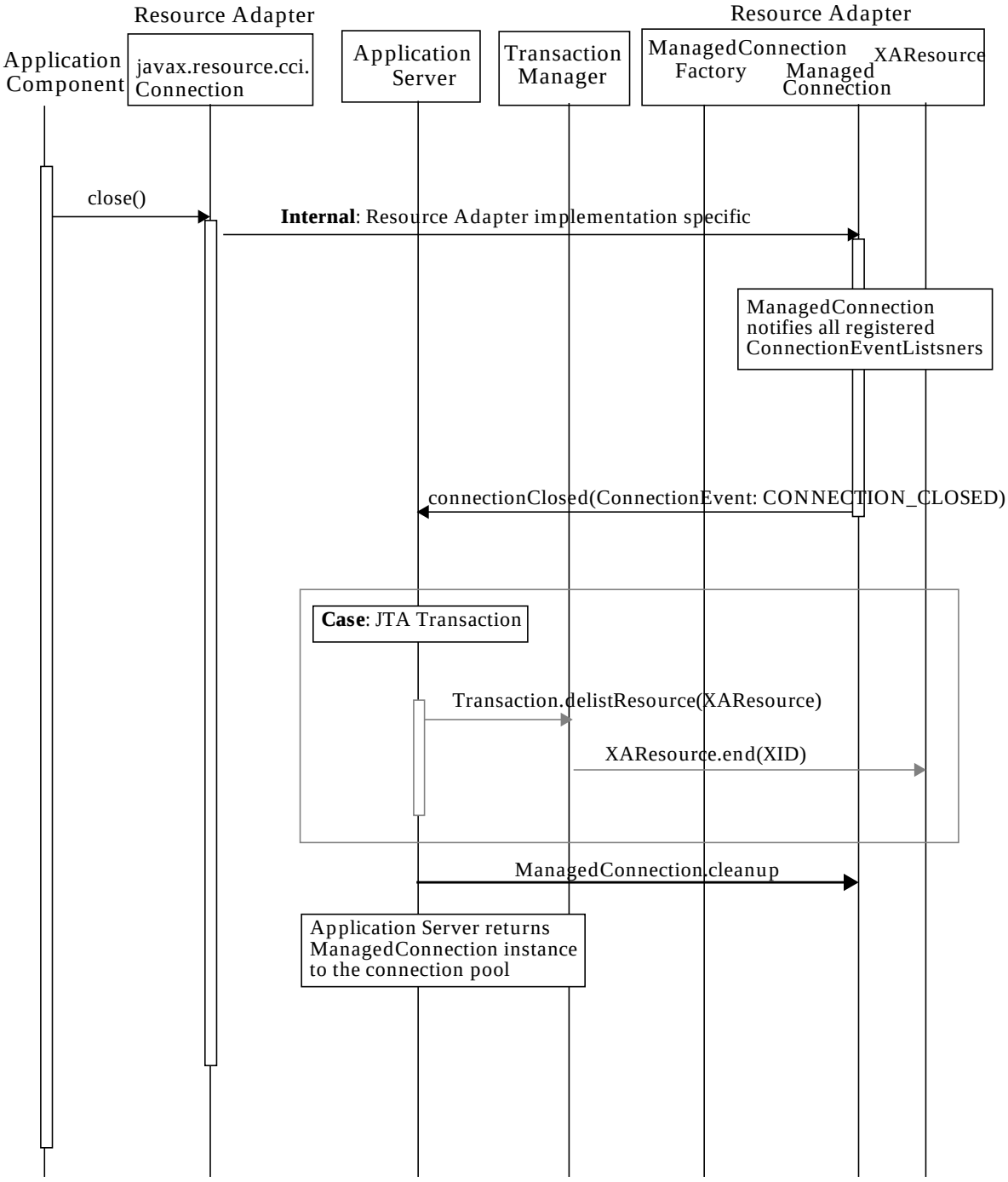
The application server can also initiate cleanup of a `ManagedConnection` instance when the container terminates the application component instance that has the corresponding connection handle. The application server should call `ManagedConnection.cleanup` to initiate the connection cleanup. After the cleanup, the application server puts the `ManagedConnection` instance into the pool to serve future allocation requests.

#### Connection Destroy

To manage the size of the connection pool, the application server can call `ManagedConnection.destroy` method to destroy a `ManagedConnection`. A `ManagedConnection` instance handles this method call by closing the physical connection to the EIS instance and releasing all system resources held by this instance.

The application server also calls `ManagedConnection.destroy` when it receives a connection error event notification that signals a fatal error on the physical connection.

**FIGURE 11.0** OID: Connection Event Notification



## 5.9 Architecture: Non-managed Environment

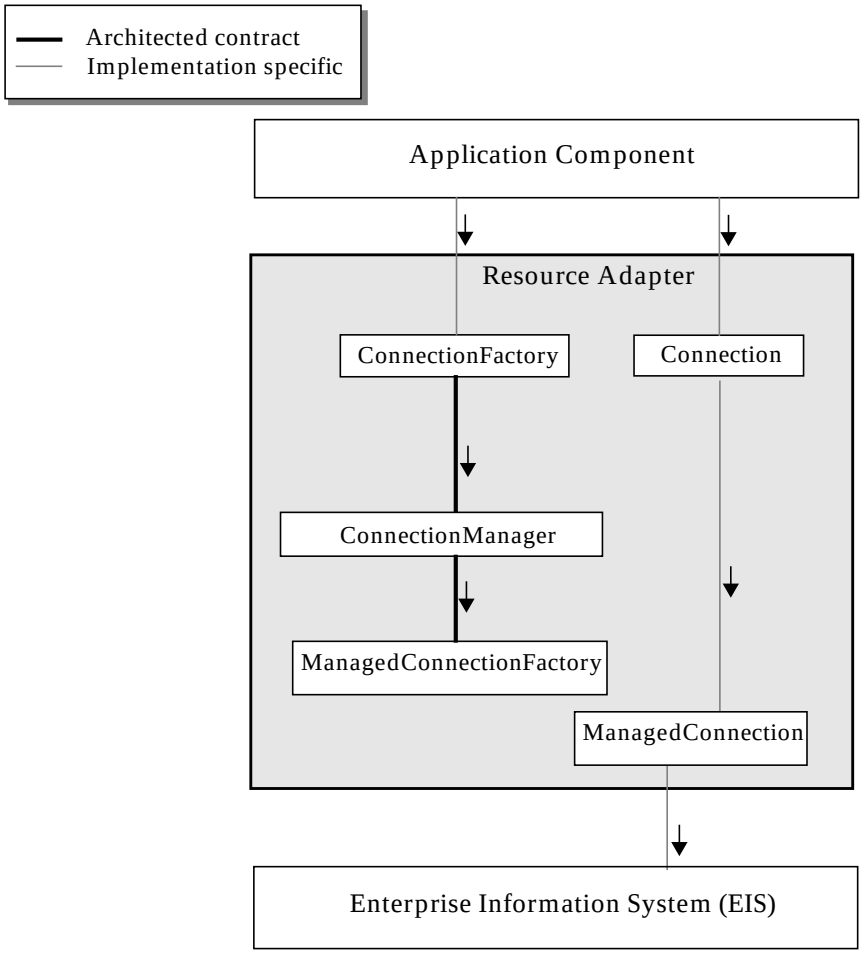
The connection management contract enables a resource adapter to be used in a two-tier application directly from an application client.

In a non-managed application scenario, the `ConnectionManager` implementation class may be provided either by a resource adapter (as a default `ConnectionManager` implementation) or by application developers. In both cases, third party vendors may provide QoS as components. Note that a default implementation of the `ConnectionManager` should be defined for a resource adapter (in terms of the functionality provided and third-party components added) only at development time.

The default `ConnectionManager` instance interposes on the connection request and delegates the request to the `ManagedConnectionFactory` instance. The `ManagedConnectionFactory` creates a physical connection (represented by a `ManagedConnection` instance) to the underlying EIS. The `ConnectionManager` gets a connection handle (of type `javax.resource.cci.Connection` for CCI) from the `ManagedConnection` and returns it to the connection factory. The connection factory returns the connection handle to the application.

A resource adapter supports interactions (shown as light shaded lines in Figure 12.0) between its internal objects in an implementation-specific way. For example, a resource adapter can use the connection event listening mechanism as part of its `ManagedConnection` implementation for connection management. However, the resource adapter is not required to use the connection event mechanism to drive its internal interactions.

**FIGURE 12.0** Architecture Diagram: Non-Managed application scenario




### 5.9.1 Scenario: Programmatic Access to ConnectionFactory

To maintain the consistency of the application programming model across both managed and non-managed environments, application code should use the `JNDI` namespace to look-up a connection factory instance.

The following code extract shows how an application client accesses a connection factory instance in a non-managed environment. The code extract does not show the use of `JNDI`. It is used as an example to illustrate the use of `ManagedConnectionFactory` and `ConnectionFactory` interfaces in the application code. Refer to section 10.5 for details on JNDI configuration and lookup.

```
// Application Client Code
// Create an instance of ManagedConnectionFactory implementation class
// passing in initialization parameters (if any) for this instance
com.myeis.ManagedConnectionFactoryImpl mcf =
     new com.myeis.ManagedConnectionFactoryImpl(...);

// Set properties on the ManagedConnectionFactory instance
// Note: Properties are defined on the implementation class and not on the
// javax.resource.spi.ManagedConnectionFactory interface
mcf.setServerName(...);
```

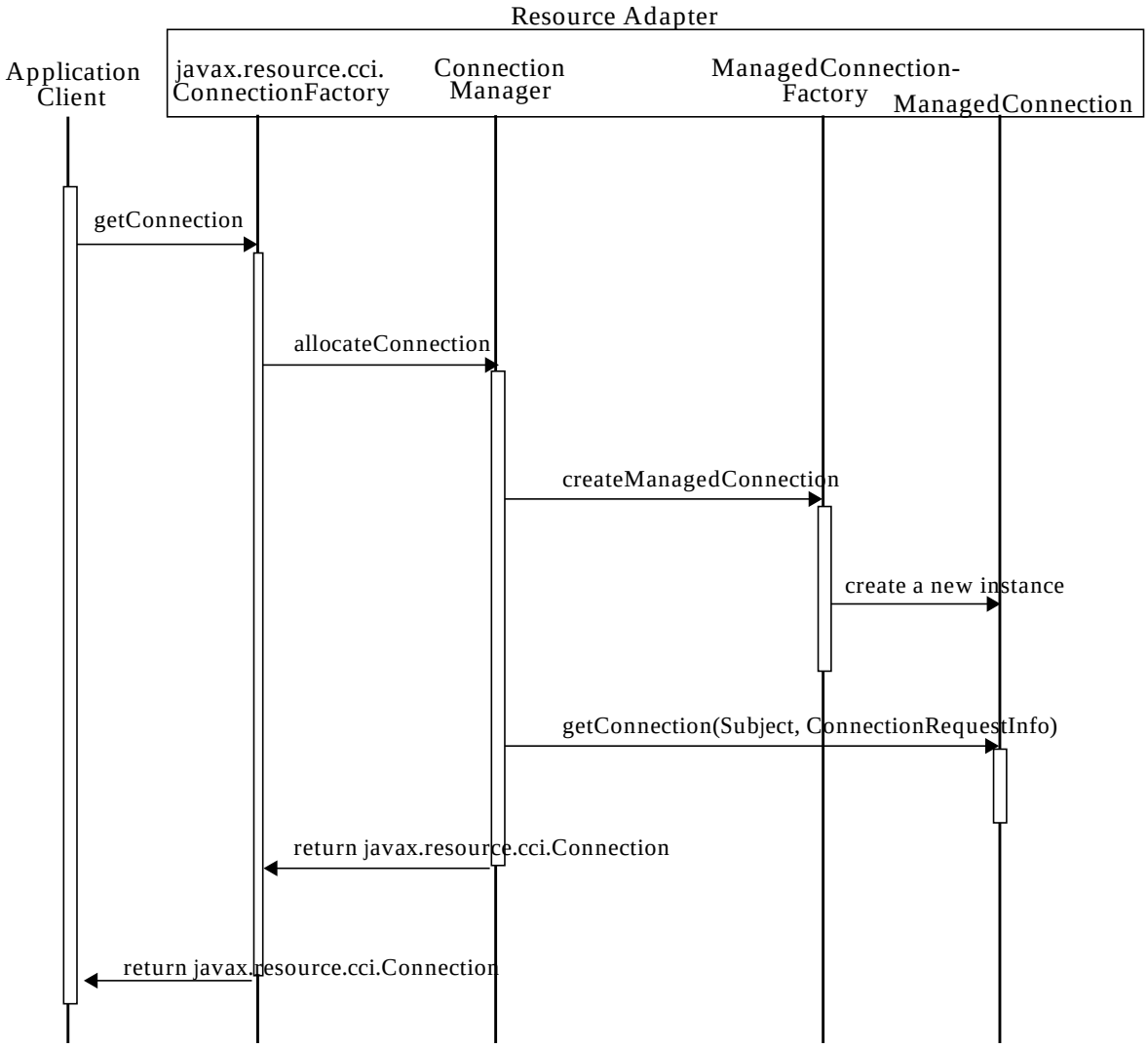
```
mcf.setPortNumber(...);

// ... set remaining properties

// Get access to connection factory. The ConnectionFactory instance
// gets initialized with the default ConnectionManager provided
// by the resource adapter
javax.resource.cci.ConnectionFactory cxf =
     (javax.resource.cci.ConnectionFactory)
           mcf.createConnectionFactory();

// Get a connection using the ConnectionFactory instance
javax.resource.cci.Connection cx = cxf.getConnection(...);

// ... use connection to access the underlying EIS instance

// Close the connection
cx.close();
```

### 5.9.2 Scenario: Connection Creation in Non-managed Application Scenario

The following object interactions are involved in the scenario shown in Figure 13.0 on page 48:

- The application client calls a method on the `javax.resource.cci.ConnectionFactory` instance (returned from the JNDI lookup) to get a connection to the underlying EIS instance.
- The `ConnectionFactory` instance delegates the connection request from the application to the default `ConnectionManager` instance. The resource adapter provides the default `ConnectionManager` implementation.
- The `ConnectionManager` instance creates a new physical connection to the underlying EIS instance by calling the `ManagedConnectionFactory.createManagedConnection` method.
- The `ManagedConnectionFactory` instance handles the `createManagedConnection` method by creating a new physical connection to the underlying EIS, represented by a `ManagedConnection` instance. The `ManagedConnectionFactory` uses the security information (passed as a `Subject` instance), any `ConnectionRequestInfo`, and its configured set of properties (such as port number, server name) to create a new `ManagedConnection` instance.
- The `ManagedConnectionFactory` initializes the state of the created `ManagedConnection` instance and returns it to the default `ConnectionManager` instance.
- The `ConnectionManager` instance calls the `ManagedConnection.getConnection` method to get an application-level connection handle. Calling the `getConnection` method does not necessarily create a new physical connection to the EIS instance. Calling `getConnection` produces a temporary handle that is used by an application to access the underlying physical connection. The actual underlying physical connection is represented by a `ManagedConnection` instance.
- The `ConnectionManager` instance returns the connection handle to the `ConnectionFactory` instance, which then returns the connection to the application that initiated the connection request.

**FIGURE 13.0** OID: Connection Creation in a Non-managed Application Scenario



## 5.10 Requirements

The following section outlines requirements for the connection management contract.

### 5.10.1 Resource Adapter

The requirements for a resource adapter are as follows:

- A resource adapter must provide implementations of the following interfaces:
  - `javax.resource.spi.ManagedConnectionFactory`
  - `javax.resource.spi.ManagedConnection`
  - `javax.resource.spi.ManagedConnectionMetaData`
- The `ManagedConnection` implementation provided by a resource adapter must use the following interface and classes to provide support to an application server for connection management (and transaction management, as explained later):
  - `javax.resource.spi.ConnectionEvent`
  - `javax.resource.spi.ConnectionEventListener`

  To support non-managed environments, a resource adapter is not required to use the above two interfaces to drive its internal object interactions.
- A resource adapter is required to provide support for basic error logging and tracing by implementing the following methods:
  - `ManagedConnectionFactory.set/getLogWriter`
  - `ManagedConnection.set/getLogWriter`
- A resource adapter is required to provide a default implementation of the `javax.resource.spi.ConnectionManager` interface. The implementation class comes into play when a resource adapter is used in a non-managed two-tier application scenario. In an application server-managed environment, the resource adapter should not use the default `ConnectionManager` implementation class.

  A default implementation of `ConnectionManager` enables the resource adapter to provide services specific to itself. These services can include connection pooling, error logging and tracing, and security management. The default `ConnectionManager` delegates to the `ManagedConnectionFactory` the creation of physical connections to the underlying EIS.
- In a managed environment, a resource adapter is not allowed to support its own internal connection pooling. In this case, the application server is responsible for connection pooling. However, a resource adapter may multiplex connections (one or more `ManagedConnection` instances per physical connection) over a single physical *pipe* transparent to the application server and components.

  In a non-managed two tier application scenario, a resource adapter is allowed to support connection pooling internal to the resource adapter.

### 5.10.2 Application Server

The requirements for an application server are as follows:

- An application server must use the interfaces defined in the connection management contract to use services provided by a resource adapter. These interfaces are as follows:
  - `javax.resource.spi.ManagedConnectionFactory`
  - `javax.resource.spi.ManagedConnection`
  - `javax.resource.spi.ManagedConnectionMetaData`

- An application server is required to provide an implementation of the `javax.resource.spi.ConnectionManager` interface. This implementation should not be specific to any particular type of resource adapter, EIS, or connection factory interface.
- An application server is required to implement the `javax.resource.spi.-ConnectionEventListener` interface and to register `ConnectionEventListener` with resource adapter to get connection-related event notifications. An application server uses these event notifications to do its pool management, transaction management, and connection cleanup.
- An application server is required to use the following interfaces (supported by the resource adapter) to provide basic error logging and tracing for its configured set of resource adapters:
    - `ManagedConnectionFactory.set/getLogWriter`
    - `ManagedConnection.set/getLogWriter`
- An application server is required to use the `javax.resource.spi.ConnectionManager` *hook-in* mechanism to provide its specific quality of services. The connector architecture does not specify the set of services the application server provides, nor does it specify how the application server implements these services.

# 6 Transaction Management

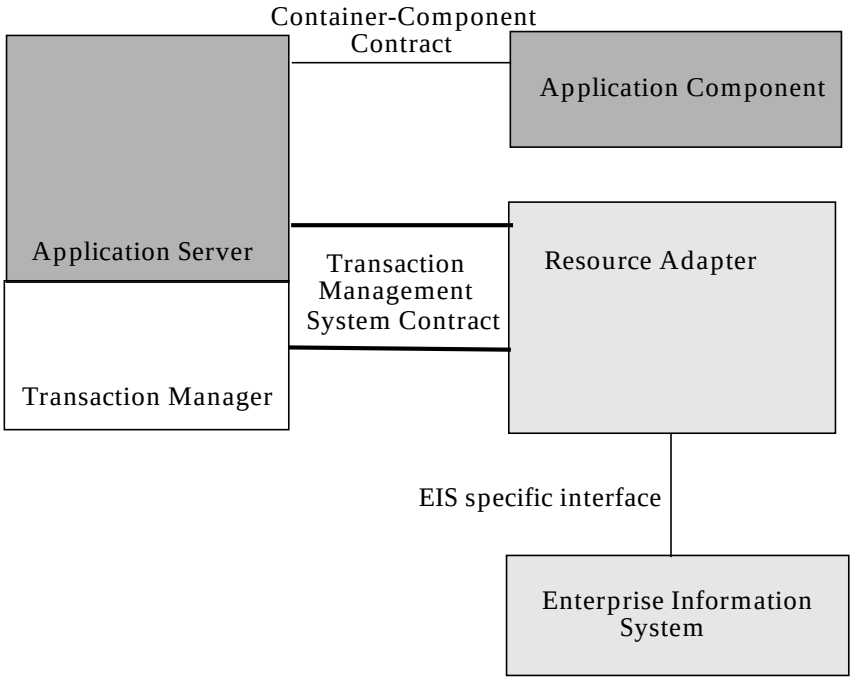
This chapter specifies the transaction management contract between an application server (and supported transaction manager) and an EIS resource manager.

This chapter focuses only on the system-level aspects of transaction management. The J2EE component model specifications describe the application level transaction model. For example, the EJB specification [1] specifies the transaction model for EJB components.

## 6.1 Overview

Figure 14.0 shows an application component deployed in a container provided by an application server. The application component performs transactional access to multiple resource managers. The application server uses a transaction manager that takes the responsibility of managing transactions across multiple resource managers.

**FIGURE 14.0** Transaction Management Contract




A resource manager can support two types of transactions:

- A transaction that is controlled and coordinated by a transaction manager external to the resource manager. This document refers to such a transaction as JTA or XA transaction.

- A transaction that is managed internal to a resource manager. The coordination of such transactions involves no external transaction managers. This document refers to such transactions as RM local transactions (or local transactions).

A transaction manager coordinates transactions across multiple resource managers. It also provides additional low-level services that enable transactional context to be propagated across systems. The services provided by a transaction manager are not visible directly to the application components.

The connector architecture defines a transaction management contract between an application server and a resource adapter (and its underlying resource manager). The transaction management contract has two parts, depending on the type of transaction:

- a JTA `javax.transaction.xa.XAResource` based contract between a transaction manager and a resource manager
- a local transaction management contract

These contracts enable an application server to provide the infrastructure and runtime environment for transaction management. Application components rely on this transaction infrastructure to support their component-level transaction model.

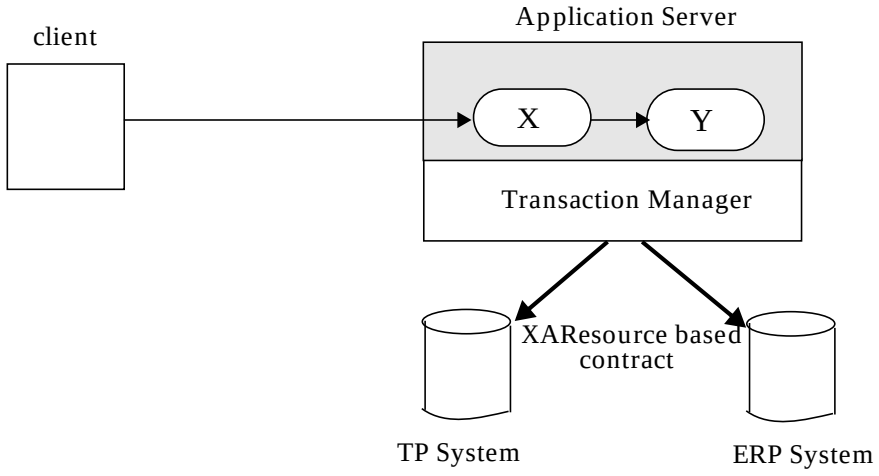
## 6.2 Transaction Management Scenarios

The following section uses a set of scenarios to present an overview of the transaction management architecture.

### 6.2.1 Transactions across multiple Resource Managers

In Figure 15.0, an application client invokes EJB component X. EJB X accesses transaction programs managed by a TP system and calls EJB Y to access an ERP system.

**FIGURE 15.0** Scenario: Transactions across multiple Resource Managers




The application server uses a transaction manager to support a transaction management infrastructure that enables an application component to perform transactional access across multiple EIS resource managers. The transaction manager manages transactions across multiple resource managers and supports propagation of the transaction context across distributed systems.

The transaction manager supports a JTA `XAResource`-based transaction management contract with a resource adapter and its underlying resource manager. The ERP system supports JTA transactions by implementing a `XAResource` interface through its resource adapter. The TP system also implements a `XAResource` interface. This interface enables the two resource managers to participate in transactions that are coordinated by an external transaction manager. The transaction manager uses the `XAResource` interface to manage transactions across the two underlying resource managers.

The EJBs X and Y access the ERP and TP system using the respective client access API for the two systems. Behind the scenes, the application server enlists the connections to both systems (obtained from their respective resource adapters) as part of the transaction. When the transaction commits, the transaction manager perform a two-phase commit protocol across the two resource managers, ensuring that all read / write access to resources managed by both TP system and ERP system is either entirely committed or entirely rolled back.

### 6.2.2 Local Transaction Management

The transactions are demarcated either by the container (called container-managed demarcation) or by a component (called component-managed demarcation). In component-managed demarcation, an application component can use the JTA `UserTransaction` interface or a transaction demarcation API specific to an EIS (for example, JDBC transaction demarcation using `java.sql.Connection`).

The EJB specification requires an EJB container to support both container-managed and component-managed transaction demarcation models. The JSP and servlet specifications require a web container to support component-managed transaction demarcation.

If multiple resource managers participate in a transaction, the EJB container uses a transaction manager to coordinate the transaction. The contract between the transaction manager and resource manager is defined using the `XAResource` interface.

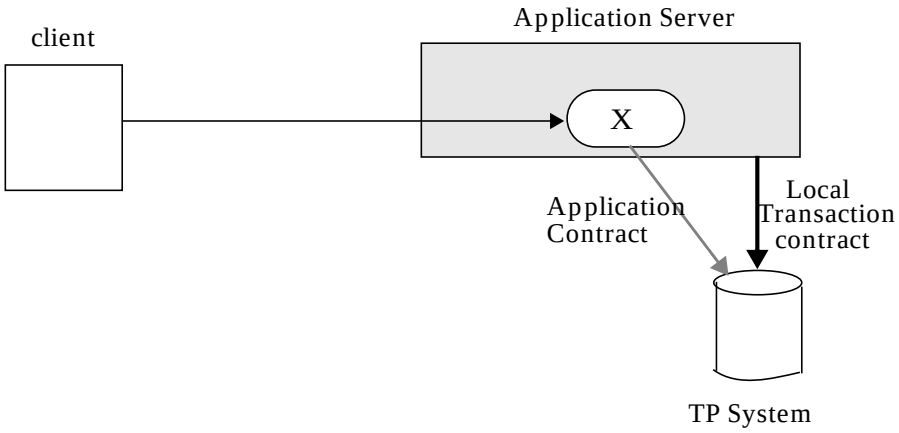
If a single resource manager instance participates in a transaction (either component-managed or container-managed), the container has two choices:

- It uses the transaction manager to manage this transaction. The transaction manager uses one-phase commit-optimization [this is described later] to coordinate the transaction for this single resource manager instance.
- The container lets the resource manager coordinate this transaction internally without involving an external transaction manager.

If an application accesses a single resource manager using a XA transaction, it has a performance overhead compared to using a local transaction. The overhead is due to the involvement of an external transaction manager in the coordination of the XA transaction.

To avoid the overhead of using a XA transaction in a single resource manager scenario, the application server may optimize this scenario by using a local transaction instead of a XA transaction. This scenario is shown in Figure 16.0.

**FIGURE 16.0** Scenario: Local Transaction on a single Resource Manager

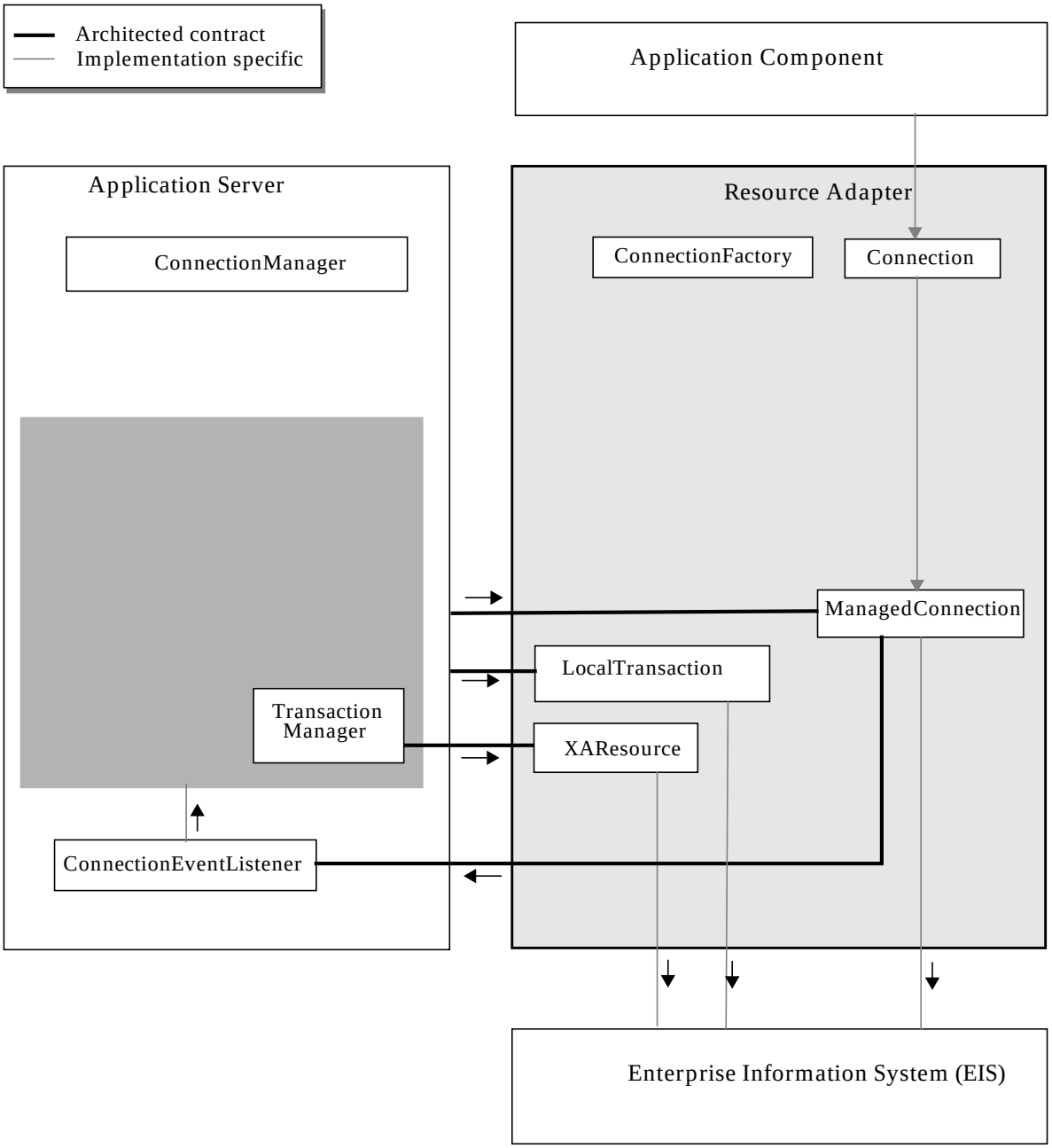



## 6.3 Transaction Management Contract

This section specifies the transaction management contract. The transaction management contract builds on the connection management contract specified in Chapter 5.

Figure 17.0 shows the interfaces and flows in the transaction management contract. It does not show the interfaces, classes, and flows that are the same in the connection management contract.

**FIGURE 17.0** Architecture Diagram: Transaction Management




### 6.3.1 Interface: ManagedConnection

The `javax.resource.spi.ManagedConnection` instance represents a physical connection to an EIS and acts as a factory of connection handles.

The following code extract shows the methods on the `ManagedConnection` interface that are defined specifically for the transaction management contract:

```
public interface javax.resource.spi.ManagedConnection {
     public XAResource getXAResource() throws ResourceException;
```

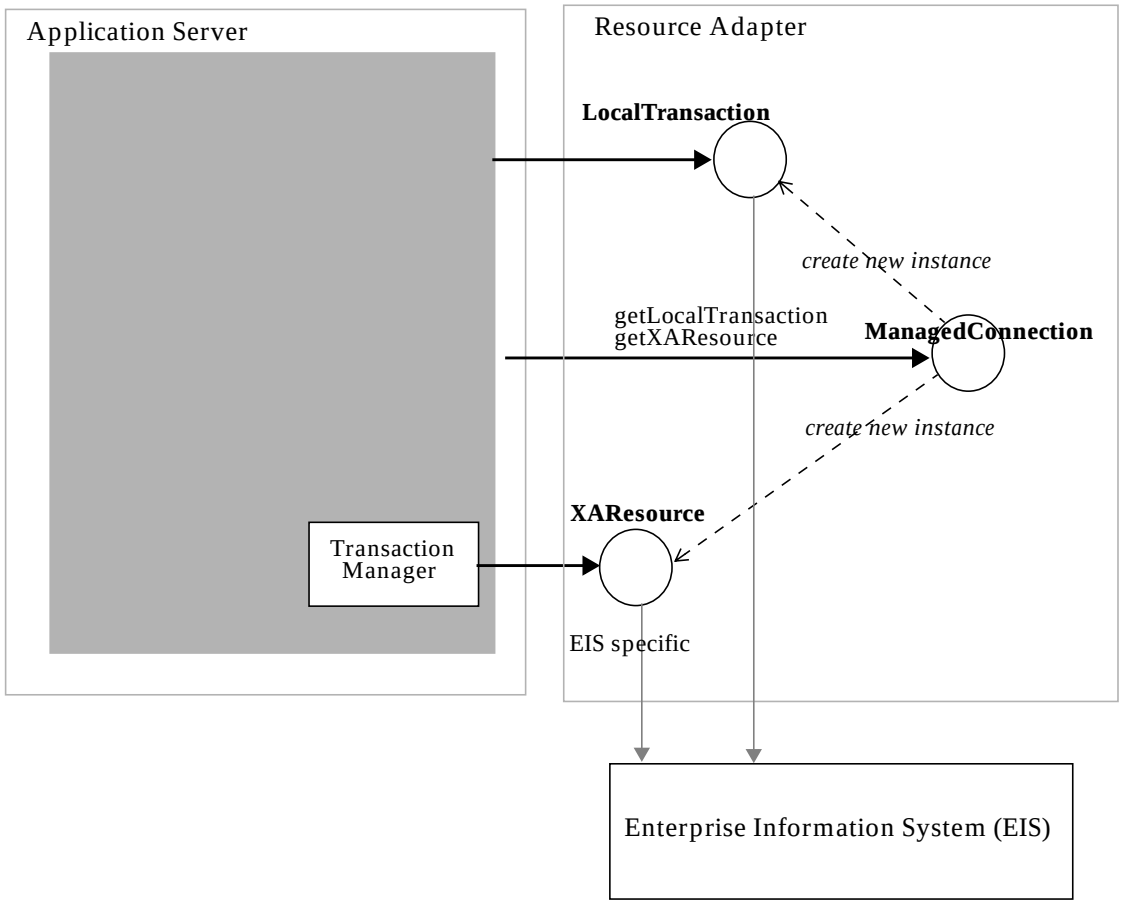
```
        public LocalTransaction getLocalTransaction()
                                        throws ResourceException;
        ...
}
```

A `ManagedConnection` instance provides access to a pair of interfaces: `javax.transaction.xa.XAResource` and `javax.resource.spi.LocalTransaction`.

Depending on the transaction support level of a resource adapter, these methods should raise appropriate exceptions. For example, if the transaction support level for a resource adapter is `NoTransaction`, an invocation of `getXAResource` method should throw a `ResourceException`. Refer to chapter 12 for details on the exception hierarchy.

Figure 18.0 illustrates this concept:

**FIGURE 18.0** ManagedConnection Interface for Transaction Management




The transaction manager uses the `XAResource` interface to associate and dissociate a transaction with the underlying EIS resource manager instance and to perform a two-phase commit protocol. The transaction manager does not directly use the `ManagedConnection` interface. The next section describes the `XAResource` interface in more detail.

The application server uses the `LocalTransaction` interface to manage local transactions.

### 6.3.2 Interface: XAResource

The `javax.transaction.xa.XAResource` interface is a Java mapping of the industry standard XA interface based on `X/Open CAE specification` [4].

The following code extract shows the interface specification for the XAResource interface. For more details and the javadocs, refer to the JTA and XA specifications:

```
public interface javax.transaction.xa.XAResource {
     public void commit(Xid xid, boolean onePhase) throws XAException;
     public void end(Xid xid, int flags) throws XAException;
     public void forget(Xid xid) throws XAException;
     public int prepare(Xid xid) throws XAException;
     public Xid[] recover(int flag) throws XAException;
     public void rollback(Xid xid) throws XAException;
     public void start(Xid xid, int flags) throws XAException;
}
```

**Implementation**

A resource adapter for an EIS resource manager implements the XAResource interface. This interface enables the resource manager to participate in transactions that are controlled and coordinated by an external transaction manager. The transaction manager uses the XAResource interface to communicate transaction association, completion, and recovery to the resource manager.

A resource adapter typically implements the XAResource interface using a low-level library available for the underlying EIS resource manager. This low-level library either supports a native implementation of the XA interface or provides a proprietary vendor-specific interface for transaction management.

A resource adapter is responsible for maintaining a 1-1 relationship between the ManagedConnection and XAResource instances. Each time a ManagedConnection.getXAResource method is called, the same XAResource instance has to be returned.

A transaction manager can use any XAResource instance (if it refers to the proper resource manager instance) to initiate transaction completion. The XAResource instance used during the transaction completion process need not be the one initially enlisted with the transaction manager for this transaction.

### 6.3.3 Interface: LocalTransaction

The following code extract shows the javax.resource.spi.LocalTransaction interface:

```
public interface javax.resource.spi.LocalTransaction {
     public void begin() throws ResourceException;
     public void commit() throws ResourceException;
     public void rollback() throws ResourceException;
}
```

A resource adapter implements the LocalTransaction interface to provide support for local transactions that are performed on the underlying resource manager. An application server uses the LocalTransaction interface to manage local transactions for a resource manager.

A later section specifies more details on the local transaction management contract.

## 6.4 Relationship to JTA and JTS

The Java™ Transaction API (JTA) [2] is a specification of interfaces between a transaction manager and the other parties involved in a distributed transaction processing system: application programs, resource managers, and an application server.

The Java Transaction Service (JTS) API is a Java binding of the CORBA Object Transaction Service (OTS) 1.1 specification. JTS provides transaction interoperability using the standard IIOP protocol for transaction propagation between servers. The JTS API is intended for vendors who implement transaction processing infrastructure for the enterprise middleware. For example, an application server vendor can use a JTS implementation as the underlying transaction manager.

### JTA Interfaces

The application server uses the javax.transaction.TransactionManager and javax.transaction.Transaction interfaces (specified in the JTA specification) for its contract with the transaction manager.

The application server uses the javax.transaction.TransactionManager interface to control the transaction boundaries on behalf of the application components that are being managed by the application server. For example, an EJB container manages the transaction states for transactional EJB components. The EJB container uses the TransactionManager interface to demarcate transaction boundaries based on the calling thread's transaction context.

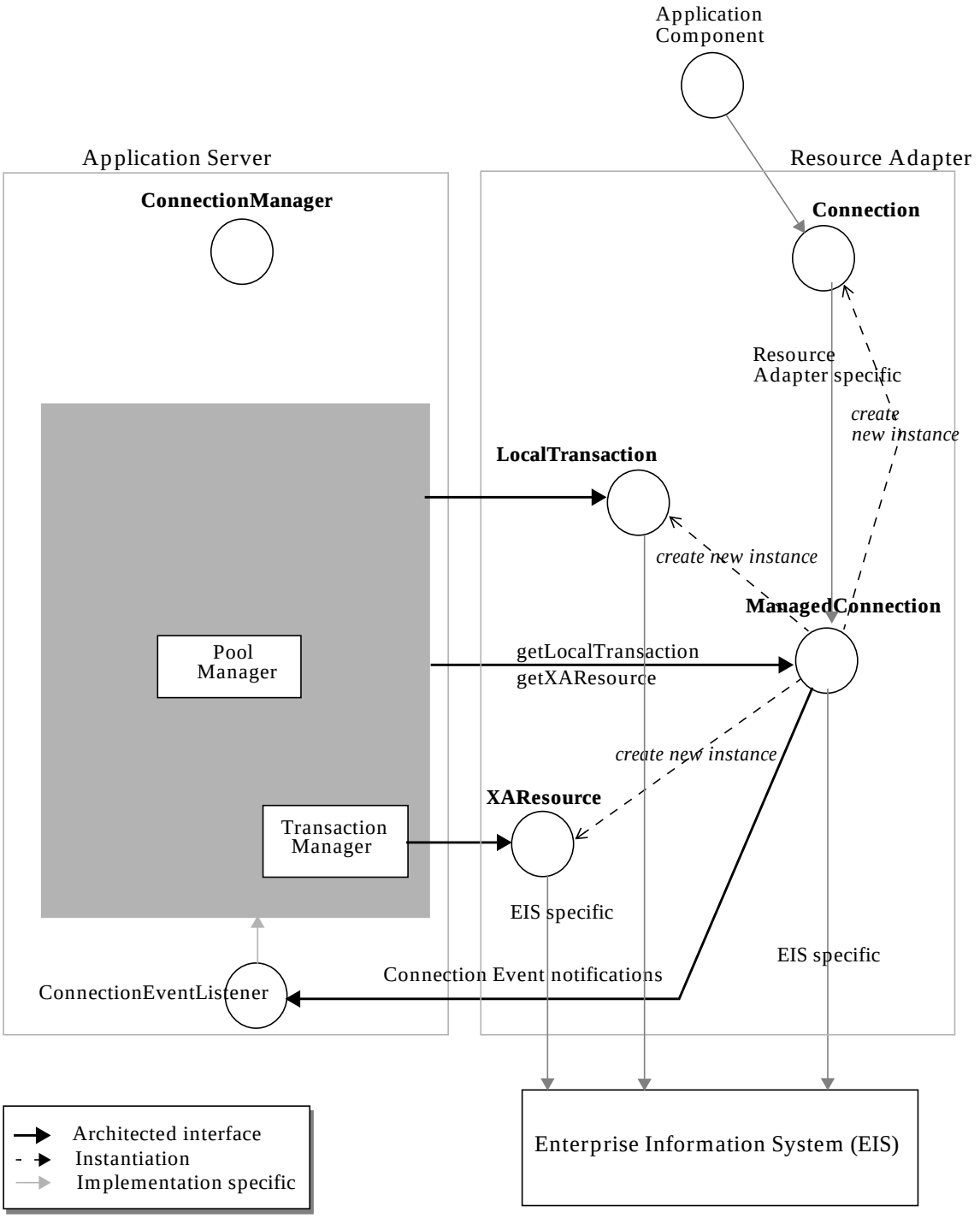
The application server also uses the javax.transaction.Transaction interface to enlist and delist transactional connections with the transaction manager. This enables the transaction manager to coordinate transactional work performed by all enlisted resource managers within a transaction.

## 6.5 Object Diagram

Figure 19.0 shows the object instances and their interactions related to transaction management.

Since the transaction management contract builds upon the connection management contract, the following diagram does not show object interactions that have been already discussed as part of Chapter 5.

**FIGURE 19.0** Object Diagram: Transaction Management



## 6.6 XAResource-based Transaction Contract

The following section specifies detailed requirements for a resource manager and a transaction manager for the `XAResource`-based transaction management contract. In this section, the following abbreviations are used: RM (Resource Manager), TM (Transaction Manager), 1PC (one phase commit protocol), and 2PC (two phase commit protocol).

### 6.6.1 Scenarios Supported

The following table specifies various transaction management scenarios and mentions whether these scenarios are within the scope of the connector architecture.

**Table 1: Transaction Management Scenarios**

| Description | Supported / NotSupported |
|---|---|
| TM does two-phase commit (2PC) on RMs that support two phase commit (as defined in RM's requirements for `XAResource` implementation in the subsection below)<br><br>**Examples of RM**: Oracle and DB2 that support 2PC in their `XAResource` implementations. | Supported based on TM's requirement to be JTA / JTS and X / Open compliant and RM's support for 2PC in `XAResource` interface. |
| TM does one-phase commit (1PC) optimization on the only RM involved in a transaction. RM supports 2PC in its `XAResource` implementation (as defined in RM's requirements for `XAResource` implementation in the subsection below).<br><br>**Example of RM**: DB2 that supports 2PC in its `XAResource` implementation. | Supported based on TM's requirement to be JTA / JTS and X / Open compliant and RM's support for `XAResource` interface.<br><br>**Note**: This scenario will also work if TM does 2PC on RM. |
| TM does one-phase commit optimization on the only RM involved in a transaction. RM does not support 2PC but supports 1PC in its `XAResource` implementation.<br><br>**Example of RM**: ERP system or mainframe TP system that does not support 2PC, but implements 1PC in its XAResource implementation as defined in the RM's requirements for 1PC. | Supported by requiring that TM must support 1PC optimization. A successful transaction coordination of 1PC only RM comes as a result of required 1PC optimization for a TM.<br><br>The rationale behind this requirement is that this scenario will be an important scenario to support for the connector architecture. |
| TM does last-resource commit optimization across multiple RMs involved in a transaction—RMs that support 2PC (for example: Oracle and DB2) and single RM that supports only 1PC (for example: ERP system). | Out of scope of the connector architecture specification |

**Table 1: Transaction Management Scenarios**

| Description | Supported / NotSupported |
|---|---|
| More than one RM that support only 1PC involved in a transaction with none or multiple 2PC enabled RMs | Out of scope of the connector architecture specification |

### 6.6.2 Resource Adapter Requirements

The connector architecture does not require that all resource adapters must support JTA `XAResource` based transaction contract.

If a resource adapter decides to support a `XAResource` based contract, then the connector architecture places certain requirements (shown below) on a resource adapter and its underlying resource manager (RM).

The following requirements refer to a resource adapter and its resource manager together as a resource manager (RM). The division of responsibility between a resource adapter and its underlying resource manager for supporting the transaction contract is implementation specific and is out of the scope of the connector architecture.

These requirements assume that a transaction manager (TM) supports JTA/XA and JTS requirements.

The following set of requirements are based on the JTA and XA specifications and should be read in conjunction with these specifications. These detailed requirements are included in this document to clearly specify the requirements from the connector architecture perspective.

#### General

- If RM supports a `XAResource` contract, then it is required to support the one-phase commit protocol by implementing `XAResource.commit` when the boolean flag `onePhase` is set to `True`. The RM is not required to implement the two-phase commit protocol support in its `XAResource` implementation.
- However, if RM supports the two-phase commit protocol, then RM is required to use the `XAResource` interface for supporting the two-phase commit protocol. Refer to the following subsection on two-phase commit for detailed requirements.
- RM is allowed to combine the implementation of 2PC protocol with 1PC optimization by implementing `XAResource.commit(onePhase=True)` in addition to the implementation requirements for 2PC.

#### One-phase Commit

- RM should allow `XAResource.commit (onePhase=True)` even if it has not received `XAResource.prepare` for the transaction branch.
- If the RM fails to commit transaction during 1PC commit, then RM should throw one of `XA_RB*` exceptions. In the exception case, RM should roll back the transaction branch's work and release all held RM resources.
- RM is responsible for deciding the outcome of a transaction branch on a `XAResource.commit` method. RM can discard knowledge of the transaction branch once it returns from the `commit` call.
- RM is not required to maintain knowledge of transaction branches to support failure recovery for the TM.
- If a `XAResource.prepare` method is called on a RM that supports only one-phase commit, then the RM should throw an `XAException` with `XAER_PROTO or XA_RB* flag`.

- RM should return an empty list of XIDs for `XAResource.recover`, because the RM is not required to maintain stable knowledge about transaction branches.

### Two-phase Commit

- If RM supports 2PC, then its implementation of 2PC is required to be compliant with 2PC protocol definition with presumed rollback as specified in the OSI DTP specification.
- RM must implement `XAResource.prepare` method and must be able to report whether it can guarantee its ability to commit the transaction branch. If RM reports that it can, RM is required to hold and record (in a stable way) all the resources necessary to commit the branch. It must hold all these resources until the TM directs it to commit or roll back the branch.
- An RM that reports a heuristic completion to the TM must not discard its knowledge of the transaction branch. The RM should discard its knowledge of the branch only when the TM calls `XAResource.forget`. RM is required to notify the TM of all heuristic decisions.
- On TM's `XAResource.commit` and `XAResource.rollback` calls, RM is allowed to report through `XAException` that it has heuristically completed the transaction branch. This feature is optional.

A TM supporting the OSI DTP specification uses the one-phase commit optimization by default to manage an RM that is the only resource involved in the transaction. The mechanism to identify to the TM a particular RM that only supports 1PC is beyond the scope of this specification.

### Transaction Association and Calling Protocol

- The RM `XAResource` implementation is required to support `XAResource.start` and `XAResource.end` for association and disassociation of a transaction (as represented by unique XID) with recoverable units of work being done on the RM.
- RM must ensure that TM invokes `XAResource` calls in the legal sequence, and must return `XAER_PROTO` or other suitable error if the caller TM violates the state tables (as defined in Chapter 6 of the XA specification (refer [4]).

### Unilateral Roll-back

- RM need not wait for global transaction completion to report an error. RM can return rollback-only flag as a result of any `XAResource.start` or `XAResource.end` call. This can happen anytime except after a successful `prepare`.
- RM is allowed to unilaterally rollback and forget a transaction branch any time before it prepares it.

### Read-Only Optimization

- Support for Read-only optimization is optional for RM implementation. An RM can respond to TM's request to prepare by asserting that the RM was not asked to update shared resources in this transaction branch. This response concludes RM's involvement in the transaction and RM can release all resources and discard its knowledge of the transaction.

### XID Support

- RM must accept XIDs from TMs. RM is responsible for using `XID` to maintain an association between a transaction branch and recoverable units of work done by the application programs.
- RM must not alter in any way the bits associated in the data portion of an XID. For example, if an RM remotely communicates an XID, it must ensure that the data bits of the XID are not altered by the communication process.

**Support for Failure Recovery**

- A full JTA compliant `XAResource` implementation that supports 2PC is required to maintain the status of all transaction branches in which it is involved. After responding affirmatively to TM `prepare` call, an RM should not erase its knowledge of the branch or of the work done in support of the branch until it receives successfully a TM's invocation to commit or roll back the branch.
- If an RM that supports 2PC heuristically completes a branch, it should not forget a branch until TM explicitly tells it to by calling `XAResource.forget`.
- On TM's `XAResource.recover` call, an RM that supports 2PC is required to return a list of all transaction branches that it has prepared or has heuristically completed.
- When a RM recovers from its own failure, it is required to recover prepared and heuristically completed branches. It should discard its knowledge of all other branches.

### 6.6.3 Transaction Manager Requirements

The following section specifies requirements of a TM. This section assumes that TM is compliant to JTA / JTS and X / Open (refer [4]) specifications.

**Interfaces**

- TM must use the `XAResource` interface supported by an RM for transaction coordination and recovery. TM must be written to handle consistently any information or status that an RM can legally return. TM must assume that it can support RMs that have different capabilities as allowed by the RM requirements specification section—RMs that make heuristic decisions and RMs that use the read-only optimization. [Requirement derived from Section 7.3, XA specification]

**XID requirements**

- TM must generate XIDs conforming to the structure defined in section 4.2 on page 19 of the XA specification (Refer [4]). The XIDs generated must be globally unique and must adequately describe a transaction branch.

**One-phase commit Optimization**

- TM's support of one-phase commit protocol optimization is required. TM uses the 1PC optimization when the TM knows that there is only one RM registered in a transaction that is making changes to shared resources. In this optimization, the TM makes its phase 2 commit request to that RM without having made a phase 1 prepare request.
- TM is not required to record (in a stable manner) such transactions, and in some failure cases, the TM may not know the outcome of the transaction completion.

**Implementation Options**

- The support of `last-resource optimization` is an implementation-specific option for a TM. A detailed specification of TM and RM's requirements for this optimization is outside the scope of the connector architecture.

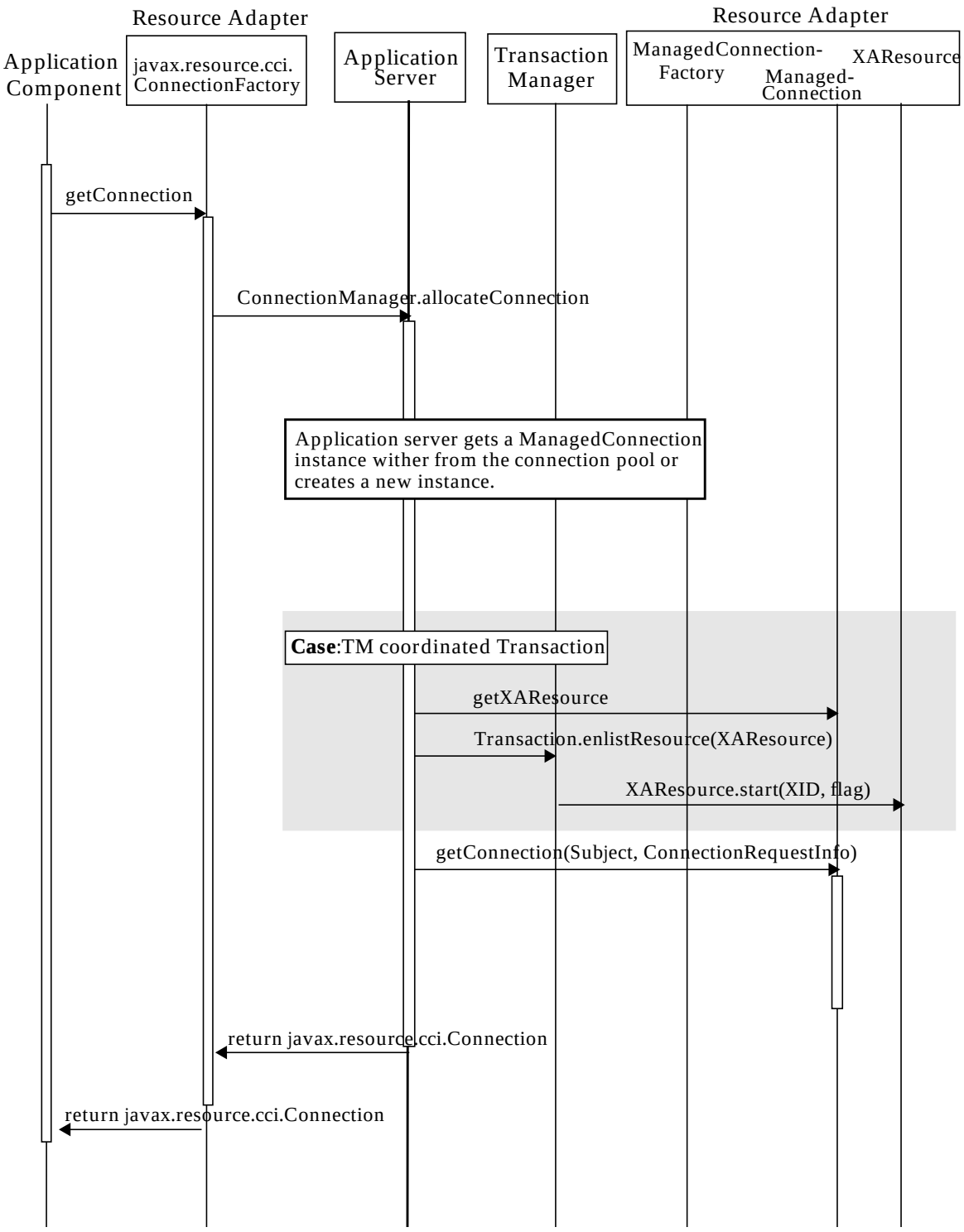
### 6.6.4 Scenario: Transactional setup for a ManagedConnection

The following object interactions are involved in the scenario shown in Figure 20.0 on page 65.

- The runtime scenario begins with a client method invocation on an EJB instance. This invocation has a transaction context (represented by a unique transaction `Xid`) associated with it if the invocation came from a client that was already participating in the transaction. Alternatively, the EJB container starts a transaction before dispatching the client request to the EJB method.

- The EJB instance calls the `getConnection` method on the `ConnectionFactory` instance. The resource adapter delegates the connection request to the application server using the connection management contract. The sequence diagram [Figure 9.0 on page 40] explains this step.
- The application server gains control and handles the connection allocation request.
- To handle the connection allocation request, the application server gets a `ManagedConnection` instance either from the connection pool or creates a new `ManagedConnection` instance. The sequence diagram [Figure 9.0 on page 40] describes this step.
- The application server registers itself as a `ConnectionEventListener` with the `ManagedConnection` instance. This enables the application server to receive notifications for various events on this connection instance. The application server uses these event notifications to manage connection pooling and transactions.
- Based on the current transaction context (associated with the connection-requesting thread and the EJB instance), the application server decides whether or not the transaction manager will participate in the coordination of the currently active transaction.
- If the application server decides that the transaction manager will manage the current transaction (the current transaction is a JTA transaction), it conducts the following transactional setup on the `ManagedConnection` instance:
    - The application server invokes the `ManagedConnection.getXAResource` method to get the `XAResource` instance associated with the `ManagedConnection` instance.
    - The application server enlists the `XAResource` instance with the transaction manager for the current transaction context. The application server uses the `Transaction.enlistResource` (specified in the JTA specification) method to enlist the `XAResource` instance with the transaction manager. This enlistment informs the transaction manager about the resource manager instance participating in the transaction.
    - The transaction manager invokes `XAresource.start` to associate the current transaction with the underlying resource manager instance. This enables the transaction manager to inform the participating resource manager that all units of work performed by the application on the underlying `ManagedConnection` instance should now be associated with this transaction.
- The application server calls the `ManagedConnection.getConnection` method to get a new application-level connection handle. The underlying physical connection is represented by a `ManagedConnection` instance.
- The application server returns the connection handle to the resource adapter. The resource adapter then passes the connection handle to the application component that had initiated the connection request.

**FIGURE 20.0** OID: Transactional setup for newly created ManagedConnection instances




### 6.6.5 Scenario: Connection Close and JTA Transactional Cleanup

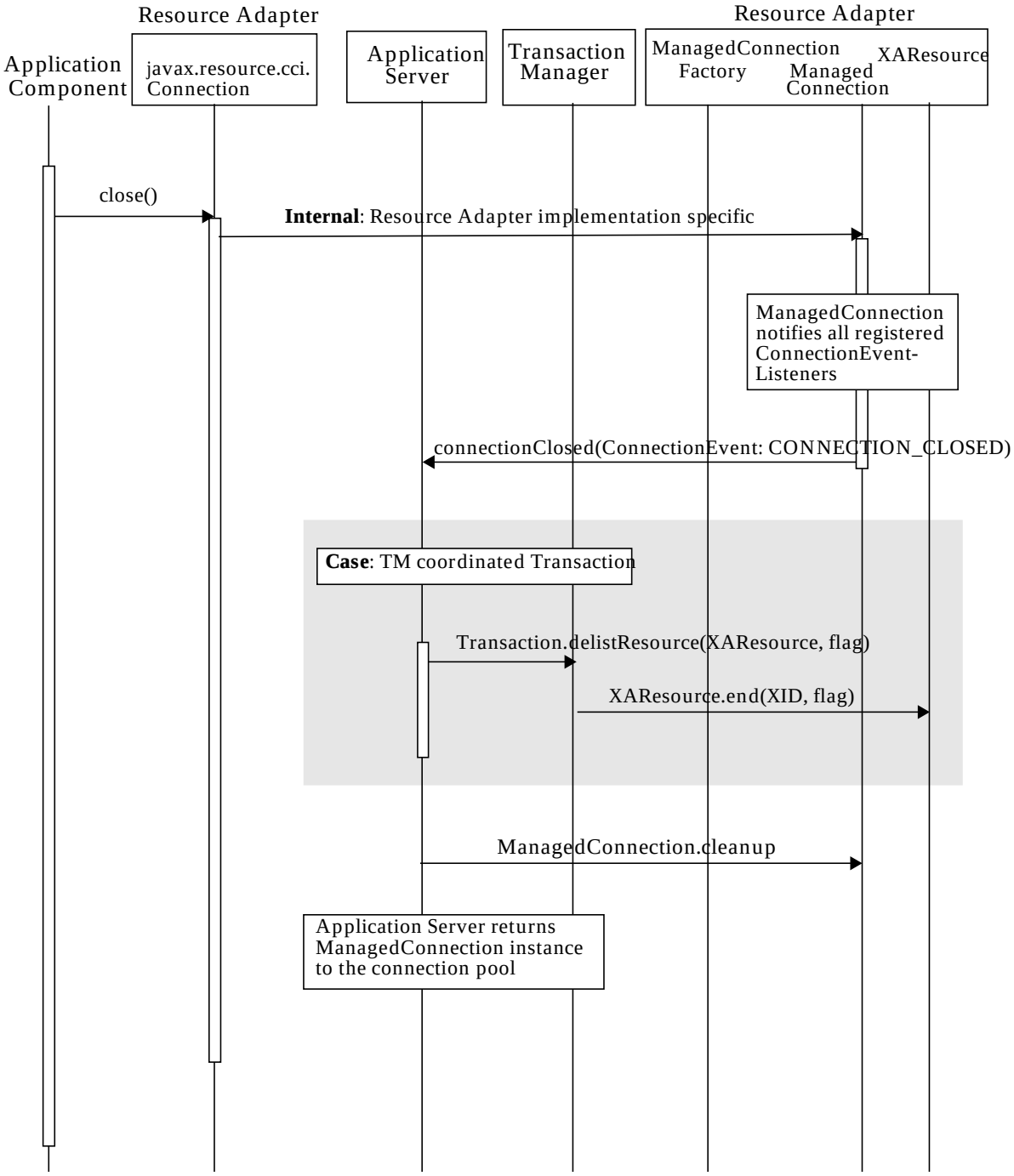
For each `ManagedConnection` instance in the pool, the application server registers a `ConnectionEventListener` instance to receive specific events on the connection. The connection

event callback mechanism enables the application server to manage connection pooling and transactions.

The scenario (shown in Figure 11.0 on page 44) involves the following steps when an application component initiates a connection close:

- The application component releases a `Connection` instance by calling the `close` method. The `Connection` instance delegates the connection close to its associated `ManagedConnection` instance. A `ManagedConnection` must not alter any state on the physical connection while handling a delegated connection close request.
- The `ManagedConnection` instance notifies all its registered listeners of the application's connection close request using the `ConnectionEventListener.connectionClosed` method. It passes a `ConnectionEvent` instance with the event type set to CONNECTION_CLOSED.
- On receiving the connection close notification, the application server performs transactional cleanup for the `ManagedConnection` instance. If the `ManagedConnection` instance was participating in a transaction manager-enlisted JTA transaction, the application server takes the following steps:
  - The application server dissociates the `XAResource` instance (corresponding to the `ManagedConnection` instance) from the transaction manager using the method `Transaction.delistResource`.
  - The transaction manager calls `XAResource.end(Xid, flag)` to inform the resource manager that any further operations on the `ManagedConnection` instance are no longer associated with the transaction (represented by the `Xid` passed in `XAResource.end` call). This method invocation dissociates the transaction from the resource manager instance.
- After the JTA transaction completes, the application server initiates a cleanup of the physical connection instance by calling `ManagedConnection.cleanup` method. After calling the method `cleanup` on the `ManagedConnection` instance, the application server returns the `ManagedConnection` instance to the connection pool.
- The application server can now use the `ManagedConnection` instance to handle future connection allocation requests from either the same or another component instance.

**FIGURE 21.0** OID: Connection Close and Transactional cleanup



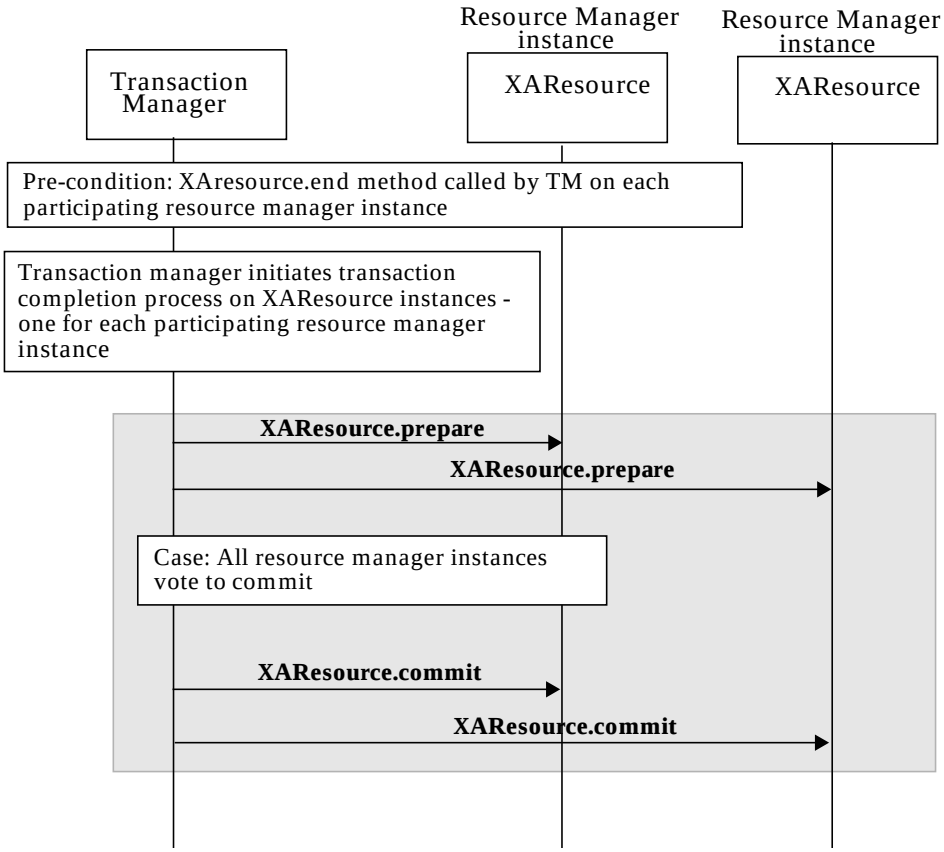

## 6.6.6 OID: Transaction Completion

The scenario in Figure 22.0 illustrates the steps taken by the transaction manager to commit a transaction across multiple resource manager instances. These steps are executed after the transaction manager calls the `XAResource.end` method for each enlisted resource manager instance.

The following steps happen in this scenario:

- The transaction manager calls `XAResource.prepare` to begin the first phase of the transaction completion protocol. The transaction manager can call any `XAResource` instance that is associated with the proper underlying resource manager instance, and is not restricted to the `XAResource` instance directly involved with the transaction initially. The application server can assume that all `XAResource` instances produced by a `ManagedConnectionFactory` instance refer to the same underlying resource manager instance.
- Assuming that all resource manager instances involved in the transaction agree to commit, the transaction manager calls `XAResource.commit` to commit the transaction. Otherwise, it calls `XAResource.rollback`.

**FIGURE 22.0** OID: Transaction Completion



## 6.7 Local Transaction Management Contract

The main motivation for defining a local transaction contract between an application server and a resource manager is to enable an application server to manage resource manager local transactions (hereafter called local transactions).

The local transaction management contract has two parts:

- The application server uses `javax.resource.spi.LocalTransaction` interface to manage local transactions transparently to an application component. The scenarios in sections 6.10 and illustrate this part of the local transaction management contract.
- The other part of the contract relates to notifications for local transaction-related events. If resource adapter supports a local transaction demarcation API (example: `javax.resource.cci.LocalTransaction` for Common Client Interface), the resource adapter needs to notify the application server of the events (transaction begin, commit, and rollback) related to the local transaction. An application server uses this part of the contract, and this is explained in section 6.8.

### 6.7.1 Interface: Local Transaction

The `javax.resource.spi.LocalTransaction` interface defines the contract between an application server and resource adapter for local transaction management. This interface is defined in section 6.3.3.

### 6.7.2 Interface: ConnectionEventListener

An application server implements the `javax.resource.spi.ConnectionEventListener` interface. It registers this listener instance with the `ManagedConnection` instance by using `ManagedConnection.addConnectionEventListener` method.

The following code extract specifies the `ConnectionEventListener` interface related to the local transaction management contract:

```
public interface javax.resource.spi.ConnectionEventListener {
     // Local Transaction Management related events
     public void localTransactionStarted(ConnectionEvent event);
     public void localTransactionCommitted(ConnectionEvent event);
     public void localTransactionRolledback(ConnectionEvent event);

     //...
}
```

The `ManagedConnection` instance notifies its registered listeners for transaction related events by calling the methods `localTransactionStarted`, `localTransactionCommitted`, `localTransactionRolledback`.

The `ConnectionEvent` class defines the following types of event notifications related to the local transaction management contract:

- LOCAL_TRANSACTION_STARTED—Notifies that a local transaction was started using the `ManagedConnection` instance.
- LOCAL_TRANSACTION_COMMITTED—Notifies that a local transaction was committed using the `ManagedConnection` instance.
- LOCAL_TRANSACTION_ROLLEDBACK—Notifies that a local transaction was rolled back using the `ManagedConnection` instance.

**Requirements**

The connector specification requires an application server to implement `ConnectionEventListener` interface and handle local transaction related events. This enables application server to achieve local transaction cleanup and transaction serial interleaving (as illustrated in the section 6.8). So the connector specification provides the necessary mechanisms for transaction management—whether these mechanisms are used in an application server depends on the application server's implementation of the transaction requirements of the J2EE component specifications.

Resource adapter is required to send local transaction events through the `ConnectionEventListener` interface when an application component starts a local transaction using application level transaction demarcation interface. An exception to this requirement is the case when transaction demarcation API supports the concept of an implicit begin of a transaction; JDBC API is an example where there is no explicit transaction begin method.

Resource adapter must not send local transaction events for local transactions managed by container.

## 6.8 Scenarios: Local Transaction Management

This section illustrates how an application server uses the event notifications from the resource adapter to manage local transactions and to restrict illegal transaction demarcations by an application component.

In these scenarios, an application component starts a local transaction using an application-level transaction demarcation interface (example: `javax.resource.cci.LocalTransaction` as defined in the CCI) supported by the resource adapter. The resource adapter, in its implementation of the transaction demarcation interface, sends event notifications related to the local transaction—namely, local transaction begin, commit, and rollback. The application server is notified of these local transaction-related events through the `ConnectionEventListener` mechanism.

### 6.8.1 Local Transaction Cleanup

A stateless session bean with bean-managed transaction demarcation starts a local transaction in a method invocation. It returns from the business method without completing the local transaction.

The application server implements the `ConnectionEventListener` interface. The resource adapter notifies the application server with `LOCAL_TRANSACTION_STARTED` event when the local transaction is started by the session bean instance.

When the session bean instance returns from the method invocation without completing the local transaction, the application server detects this as an incomplete local transaction because it has not received any matching `LOCAL_TRANSACTION_COMMITTED` or `LOCAL_TRANSACTION_ROLLEDBACK` event from the resource adapter.

On detecting an incomplete local transaction, the application server terminates the stateless session bean instance and throws an exception to the client.

### 6.8.2 Component Termination

The application server terminates a component instance—for example, because of some system exception in a method invocation.

On termination of a component instance, the application server cleans up all `ManagedConnection` instances being used by this component instance. The cleanup of a connection involves resetting all local transaction and client-specific state. This state is maintained internal to the `ManagedConnection` instance.

The application server initiates a cleanup of a `ManagedConnection` instance by calling `ManagedConnection.cleanup`. After the cleanup, the application server returns this connection to the pool to serve future allocation requests.

### 6.8.3 Transaction Interleaving

The application server uses the connection event listener mechanism (specified through the interfaces `ConnectionEventListener` and `ConnectionEvent`) to flag illegal cases of transaction demarcation. The application server implements the `ConnectionEventListener` interface to support this scenario.

The following subsection illustrates a scenario for component-managed transaction demarcation.

#### Scenario

An EJB component (with bean managed transaction demarcation) starts a local transaction using the application-level transaction demarcation interface (example: `javax.resource.cci.LocalTransaction` as defined in the CCI) supported by the resource adapter. It then calls the `UserTransaction.begin` method to start a JTA transaction before it has completed the local transaction.

In this scenario, the EJB component has started but not completed the local transaction. When the application component attempts to start a JTA transaction by invoking the `UserTransaction.begin` method, the application server detects it as a transaction demarcation error and throws an exception from the `UserTransaction.begin` method.

When the application component starts the local transaction, the resource adapter notifies the application server of the LOCAL_TRANSACTION_STARTED connection event. When the component invokes the `UserTransaction.begin` method, the application server detects an error condition, because it has not received the matching LOCAL_TRANSACTION-_COMMITTED or LOCAL_TRANSACTION_ROLLEDBACK event from the resource adapter for the currently active local transaction.

## 6.9 Connection Sharing

The following section is based on J2EE 1.3 platform specification.

When multiple connections acquired by a J2EE application use the same resource manager, containers may attempt to share connections within the same transaction scope. Sharing connections typically results in efficient usage of resources and better performance.

Connections to resource managers acquired by J2EE applications are considered potentially shared or shareable. However, a J2EE application component that intends to use a connection in an unshareable way must leave a deployment hint to that effect, which will prevent the connection from being shared by the container. Examples of unshareable usage of a connection include changing the security attributes, isolation levels, character settings, localization configuration.

Containers must not attempt to share connections that are marked `unshareable`. If a connection is marked `shareable`, it must be transparent to the application whether the connection is actually shared or not, provided that the application is properly using the connection in a shareable manner.

J2EE application components may use the optional deployment descriptor element `res-sharing-scope` to indicate whether a connection to a resource manager is shareable or unshareable. Containers should assume connections to be shareable if no deployment hint is provided. Refer to EJB specification and the Servlet specification for a description of the deployment descriptor element.

J2EE application components may cache connection objects and reuse them across multiple transactions. Containers that provide connection sharing should transparently switch such cached connection objects (at dispatch time) to point to an appropriate shared connection with the correct transaction scope. Refer section 6.11 for more details on connection association.

**Requirements**

Containers may choose to provide connection sharing, but are not required to do so. Refer to the following section for a special case of connection sharing as applied to resource adapters that support local transactions.

## 6.10 Transaction Scenarios

This section specifies requirements for various transaction scenarios.

### 6.10.1 Requirements

J2EE platform specification identifies the following as transactional resources:

- JDBC connections
- JMS sessions
- Resource adapter connections at `XATransaction` level

J2EE platform specification requires that J2EE product provider must transparently support transactions that span multiple components and transactional resources. These requirements must be met regardless of whether a J2EE product is implemented as a single process, multiple processes on the same node or multiple processes on multiple nodes.

In addition, a J2EE product provider is required to support a transactional application that is comprised of servlets or JSP pages accessing multiple enterprise beans within a single transaction. Each component may also acquire one or more connections to access transactional resources. J2EE product provider must support scenario where multiple components in an application access transactional resources as part of a single transaction.

J2EE platform specification requires J2EE platform product to support resource adapter at `XATransaction` level as a transactional resource. It must be possible to access such resource adapter from multiple application components within a single transaction.

J2EE connector architecture specifies an additional requirement that is applicable to resource adapters that support local transactions. Note that both `LocalTransaction` and `XATransaction` resource adapters support local transactions.

Application server is required to use local transaction in a scenario where the following conditions hold:

- Multiple components use a single resource adapter that is local transaction capable
- Components get connections to the same EIS instance
- Components have not specified `res-sharing-scope` flag as `unshareable`. This condition accounts for potential shareability of connections in terms of security context. client-specific connection parameters and EIS specific configuration.

Note that this requirement does not apply to a local transaction that is started by a component using an application level transaction demarcation API that is specific to a resource adapter.

Application server may use connection sharing mechanism to implement this local transaction requirement.

Application server is required to support transaction scenario where access to a non-transactional resource is combined with access to one or more transactional resource within a single transaction. For example, in a container-managed transaction, EJB accesses JDBC and JMS resources and also accesses a non-transactional EIS using its resource adapter. If there is failure

during the above scenario, transactional resource managers operating under the transaction should rollback, but the recovery of the non-transactional resource is defined unspecified in this specification.

Application server is not required to support any additional transaction scenarios beyond the above set of scenarios. A J2EE application should not depend on an application server's support for any optional transaction scenarios. Application should also not depend on whether or not container detects that a specific optional transaction scenario is illegal. Any errors in optional transaction scenarios are considered application programming errors.

### 6.10.2 Illustrative Scenarios

For purpose of illustration, the following are examples of optional transaction scenarios. The following section also describes (in a non-prescriptive manner) issues in support for these scenarios by an application server:

- Within a transaction, an EJB acquires connections to two different resource managers X and Y using respective non-XA local transaction capable resource adapters.

  The container cannot manage a local transaction across two different resource managers. Since resource adapters (and underlying resource managers) are not XA capable, container cannot use XA in this case. However, a J2EE application should not depend on the container to detect this illegal scenario.

- Within a transaction, EJB A acquires connection to a resource manager X using a non-XA local transaction capable resource adapter. Next, EJB B under the same transaction context acquires connection to a different resource manager Y using a non-XA local transaction capable resource adapter

  The container cannot manage a local transaction across two different resource managers. Since resource adapters are not XA capable, container cannot use XA in this case. However, a J2EE application should not depend on the container to detect this illegal scenario.

- Within a transaction, EJB A acquires connection to a resource manager X using non-XA local transaction capable resource adapter. Next, the same EJB (or EJB B) under the same transaction context acquires connection to a different resource manager Y using a XA capable resource adapter

  This scenario may be supported if transaction manager supports last resource commit optimization. Since this optimization feature is optional and not specified in the connector architecture, a J2EE application should not depend on support for this scenario.

- Within a transaction, EJB A acquires connection to an resource manager X using a XA capable resource adapter. Next, the same EJB (or another EJB B) under the same transaction context acquires connection to a different resource manager Y using non-XA local transaction capable resource adapter

  This scenario may be supported if transaction manager supports last resource commit optimization. Since this optimization feature is optional and not specified in the connector architecture, a J2EE application should not depend on support for this scenario.

### 6.10.3 Scenario: Local Transaction

The following scenario illustrates use of the connection sharing mechanism to implement requirement for a local transaction to span components.

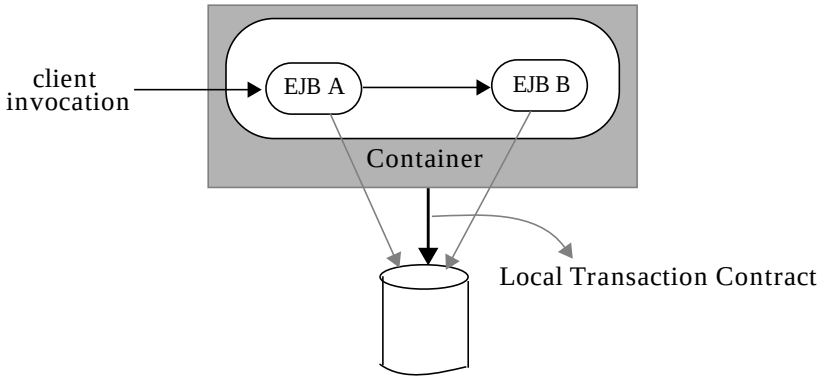
In this scenario, two EJB components get connections to the same EIS resource manager within a single transaction. Both EJBs use the same local transaction capable resource adapter.

A local transaction is associated with a single physical connection. Both EJB components in this scenario share the same physical connection under the local transaction scope. Container takes the responsibility of managing connection sharing as illustrated in the following scenario.

To share a physical connection in the local transaction scope, container assumes the connection to be shareable unless it has been marked `unshareable` in the `res-sharing-scope`. Container uses connection sharing in a manner that is transparent to application components.

In Figure 23.0, the stateful session beans A and B have container-managed transaction demarcation with the transaction attribute set to `Required`. Both EJBs A and B access a single EIS resource manager as part of their business logic.

**FIGURE 23.0** Scenario to illustrate Local Transaction Management

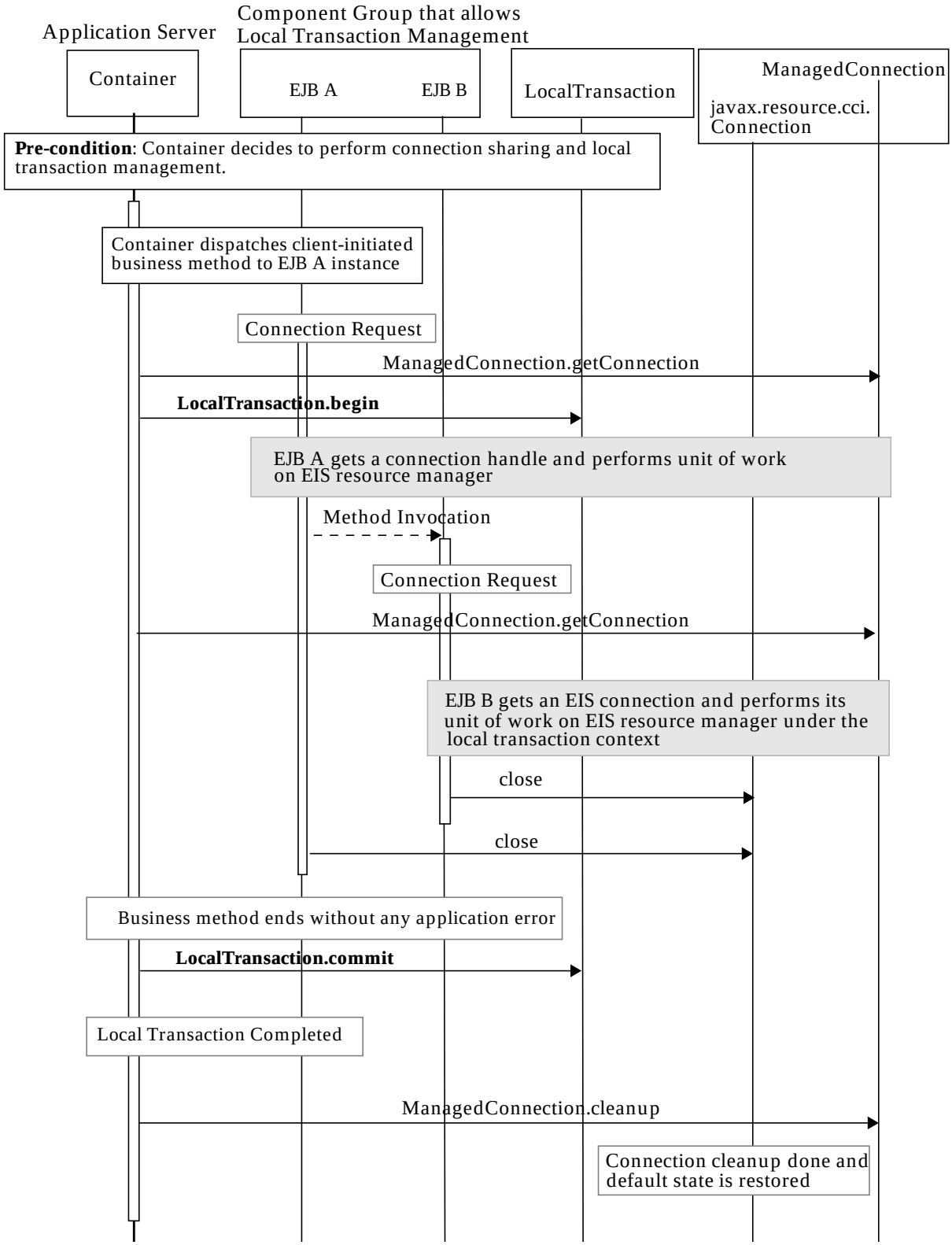



The following steps happen in this scenario:

- The client invokes a method on EJB A with no transaction context. In its method implementation, the EJB A acquires a connection to the EIS instance.
- When acquiring the connection, the container starts a local transaction by invoking `begin` method on the `javax.resource.spi.LocalTransaction` instance. The local transaction is *tied* to the `ManagedConnection` instance that is associated with the connection handle acquired by the component in the previous step.
- After the local transaction starts, any recoverable unit of work performed by the EJB A instance on the EIS resource manager (using the acquired connection) is automatically included under the local transaction context.
- EJB A now invokes a method on the EJB B instance. In this scenario, EJB A does not close the connection handle before invoking the method on EJB B instance.

  **Note**: A container should ensure that the connection sharing mechanism is equally applicable if EJB A were to close the connection handle before calling the EJB B instance.
- In the invoked method, the EJB B instance makes a request to acquire a connection to the same EIS resource manager.
- The container returns a connection handle using the same `ManagedConnection` instance that was used for handling the connection request from the EJB A instance.
- The container retains the association of the `ManagedConnection` instance with the local transaction context across the method invocation from EJB A to EJB B. This means that any unit of work that the EJB B instance will perform on the EIS resource manager (using its acquired connection handle) will be automatically included as part of the current local transaction. The connection state (for example, any open cursors) can also be retained across method invocations when the physical connection is shared.

**FIGURE 24.0** OID: Connection Sharing across Component instances



- Before the method invocation on the EJB B instance completes, EJB B calls close on the connection handle. The container should not initiate any cleanup of the physical connection at this time since there is still an uncompleted local transaction associated with the shared physical connection. In this scenario, the cleanup of a physical connection refers to dissociation of the local transaction context from the `ManagedConnection` instance.
- When EJB A regains control, EJB A can use the same connection handle (provided EJB A had not called close on the connection handle) to access EIS resources; all recoverable units of work on the EIS resource manager will be included in the existing local transaction context.

  **Note:** If EJB A closes the connection handle before calling EJB B, and then reacquires the connection handle (when regaining control), the container should ensure that the local transaction context stays associated with the shared connection**.**
- EJB A eventually calls close on its connection handle. The container gets a connection close event notification based on the scenario described in section 5.8.3.
- Since there is an uncompleted local transaction associated with the underlying physical connection, the container does not initiate a cleanup of the `ManagedConnection` on receiving the connection close event notification. The container must still go through the completion process for the local transaction.
- When the business method invocation on the EJB A instance completes successfully without any application error, the container starts the completion protocol for the local transaction. The container calls the `LocalTransaction.commit` method to commit the transaction.
- After the local transaction completes, the container initiates a cleanup (now there is no active local transaction) of the physical connection instance by calling `ManagedConnection.cleanup` method.

  **Note**: The container should initiate cleanup of the `ManagedConnection` instance in the case where EJB A does not call close on the connection handle before returning. The container identifies this need for cleanup of `ManagedConnection` based on the scope of connection sharing.
- On the `cleanup` method invocation, the `ManagedConnection` instance does a cleanup of its local transaction related state and resets itself to a default state.
- The container returns the physical connection to the pool for handling subsequent connection requests.

## 6.11 Connection Association

According to the connection management contract, a connection handle is created from a `ManagedConnection` instance using the method `ManagedConnection.getConnection`. A connection handle maintains an association with the underlying `ManagedConnection` instance.

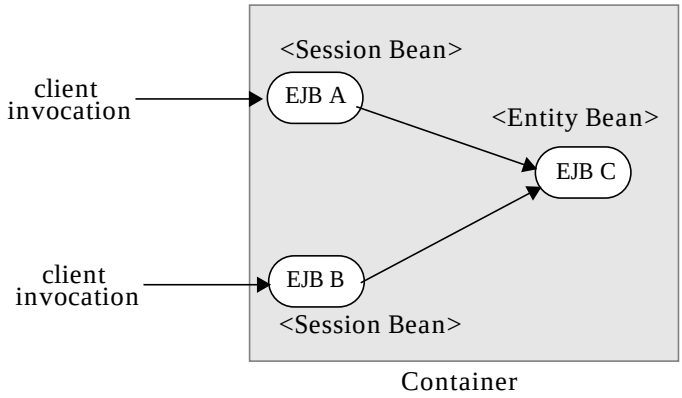
**Scenario**

In the scenario shown in Figure 25.0, session bean A acts as a client of entity bean C and makes invocations on methods of entity bean C. Another session bean B also acts as a client of entity bean C. The EJB C is an entity bean that may be shared across multiple clients.

EJBs A, B and C get connections to the same EIS. These EJBs have marked `res-sharing-scope` for these connections to be `shareable`.

EJBs A and C define a connection sharing scope. Both EJBs A and C share the same physical connection across a transaction that spans methods on EJBs A and C. Similarly, EJB B and C define another connection sharing scope. EJBs B and C also share the same physical connection across a transaction that spans two components.

**FIGURE 25.0** Connection Sharing Scenario

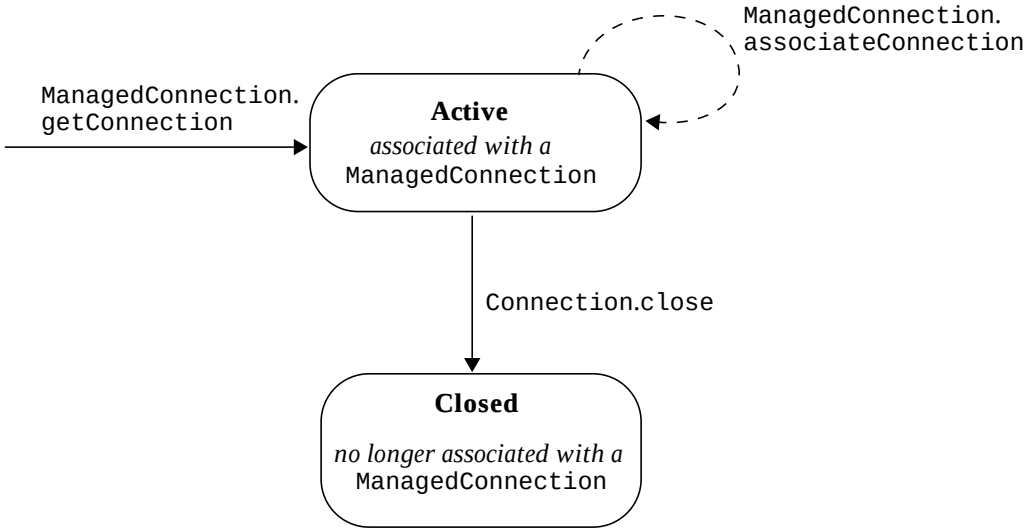



In this scenario, entity bean C instance obtains an application-level connection handle (using the method `getConnection` on the `ConnectionFactory`) during its creation. Entity bean C instance holds the connection handle during its lifetime.

EJB A instance gets a connection handle and invokes a method on EJB C. At a different instant, EJB B instance gets a connection handle and invokes a method on EJB C.

In both cases, depending on the connection sharing scope (defined in terms of the shared physical `ManagedConnection` instance) in which the EJB C instance is called, the container supports mechanism to associate the connection handle (held by the EJB C instance as part of its state) with the current `ManagedConnection` instance.

**FIGURE 26.0** State diagram of application-level Connection Handle




### Connection Association

The interface `ManagedConnection` defines method `associateConnection` as follows:

```
public interface javax.resource.spi.ManagedConnection {
     public void associateConnection(Object connection)
```

```
            throws ResourceException;
    //...
}
```

The container uses the `associateConnection` method to change the association of an application-level connection handle with a `ManagedConnection` instance. The container finds the right `ManagedConnection` instance (depending on the connection sharing scope) and calls the `associateConnection` method. To achieve this, container needs to keep track of connection handles (acquired by component instances) and `ManagedConnection` instances using an implementation specific mechanism.

The `associateConnection` method implementation for a `ManagedConnection` should dissociate the connection handle (passed as a parameter) from its currently associated `ManagedConnection` and associate the new connection handle with itself.

Note that the switching of connection association must happen only for connection handles and `ManagedConnection` instances that correspond to the same `ManagedConnectionFactory` instance. The container should enforce this restriction in an implementation-specific manner. If a container cannot enforce the restriction, then the container should not use the connection association mechanism.

**Requirements**

The container is required to provide a mechanism to change the association of a connection handle to different `ManagedConnection` instances depending on the connection sharing and transaction scope. This mechanism is used in scenarios where components hold on to connection handles across different local transaction and connection sharing scopes.

The container may use connection association mechanism in the `XAResource`-based transaction management contract.

The resource adapter is required to implement the `associateConnection` method to support connection sharing. The container makes a decision on whether or not to use the `associateConnection` method implemented by a resource adapter. The support for this method is required independent (note that the container makes the decision to invoke `associateConnection` method) of the transaction support level of the resource adapter.

## 6.12 Local Transaction Optimization

If all the work done as a part of a transaction uses a single resource manager, the application server can use local transaction in place of an externally coordinated JTA transaction. The use of local transaction avoids the overhead of initiating a global transaction (and involving TM for transaction coordination) and leads to a performance optimization.

Since a typical application accesses a single resource manager, the local transaction optimization is a useful performance optimization for transaction management.

The application server manages local transaction optimization transparent to the J2EE application. Whenever a container-managed or bean-managed transaction is started, the container may attempt local transaction optimization.

At transaction begin, a container can not determine apriori whether or not the unit of work done as part of this transaction will use a single resource manager. The container uses an implementation-specific mechanism to achieve local transaction optimization. For example, the container can choose to start a local transaction (when the first resource manager is accessed) and lazily start a JTA transaction only when more than one resource managers are accessed in an existing transaction. The mechanism through which the application server (and its transaction manager) coordinates the initial local transaction and lazily started JTA transactions is outside the scope of the connector specification. An illustrative way is to manage this case as a last-

agent optimization. Refer J2EE platform specification [8] for more details on the local transaction optimization.

### Requirements

The container is not required to support the local transaction optimization.

## 6.13 Requirements

The following section outlines requirements for the transaction management contract.

### 6.13.1 Resource Adapter

A resource adapter can be classified based on the level of transaction support, as follows:

- Level `NoTransaction`—The resource adapter supports neither resource manager local nor JTA transactions. It implements neither `XAResource` nor `LocalTransaction` interfaces.
- Level `LocalTransaction`—The resource adapter supports resource manager local transactions by implementing the `LocalTransaction` interface. The local transaction management contract is specified in the section 6.7.
- Level `XATransaction`—The resource adapter supports both resource manager local and JTA transactions by implementing `LocalTransaction` and `XAResource` interfaces respectively. The requirements for supporting `XAResource`-based contract are specified in section 6.6.

**Note**: Other levels of support (includes any transaction optimizations supported by an underlying resource manager) are outside the scope of the connector architecture.

The above levels reflect the major steps of transaction support that a resource adapter needs to make to allow external transaction coordination. Depending on its transactional capabilities and requirements of its underlying EIS, a resource adapter can choose to support any one of the above transaction support levels.

### 6.13.2 Application Server

An application server is required to support resource adapters with all three levels of transaction support—`NoTransaction`, `LocalTransaction`, and `XATransaction`.

The following are the requirements for an application server for the transaction management contract:

- The application server is required to support a transaction manager that manages transactions using the JTA `XAResource`-based contract. The requirements for a transaction manager to support an `XAResource`-based contract are specified in section 6.6.3 on page 63.
- The application server is required to use the `LocalTransaction` interface-based contract to manage local transactions for a resource manager.
- The application server is required to use the deployment descriptor mechanism to ascertain the transactional capabilities of a resource adapter. Refer to section 10.3 for details on the deployment descriptor specification.
- The application server is required to implement the `javax.resource.spi.ConnectionEventListener` interface to get transaction-related event notifications.

# 7 Security Architecture

The following chapter specifies a security architecture for the integration of EISs with the J2EE platform. It adds EIS integration-specific security details to the security requirements specified in other J2EE specifications.

## 7.1 Overview

It is critical that an enterprise be able to depend on the information in its EIS for its business activities. Any loss or inaccuracy of information or any unauthorized access to the EIS can be extremely costly to an enterprise. There are mechanisms that can be used to protect an EIS against such security threats, including:

- Identification and authentication of principals (human users) to verify they are who they claim to be.
- Authorization and access control to determine whether a principal is allowed to access an application server and / or an EIS.
- Security of communication between an application server and an EIS. Communication over insecure links can be protected using a protocol (for example, Kerberos) that provides authentication, integrity, and confidentiality services. Communication can also be protected by using a secure links protocol (for example, SSL).

## 7.2 Goals

The security architecture is designed to meet the following goals:

- Extend the end-to-end security model for J2EE-based applications to include integration with EISs based on the connector architecture.
- Support authentication and authorization of users who are accessing EISs.
- Keep the security architecture technology neutral and enable the specified security contract to be supported by various security technologies.
- Enable the security architecture to support a range of EISs with different levels of security support and existing security environments.
- Support security configuration of a resource adapter in an operational environment.
- Keep the security model for connector architecture-based EIS integration transparent to an application component provider. This includes providing support for single sign-on across multiple EISs.

The non-goals of the security model for EIS integration are as follows:

- Mandate a specific technology and describe how it can be used to implement the security architecture for connector architecture-based EIS integration.
- Specify and mandate a specific security policy. The security architecture enables an application server and EIS to support implementation and administration of security policies based on their respective requirements.

## 7.3 Terminology

The following terms have been used in this chapter:

- `Principal`: A principal is an entity that can be authenticated by an authentication mechanism deployed in an enterprise. A principal is identified using a `principal name` and authenticated using `authentication data`. The content and format of the principal name and the authentication data depend upon the authentication mechanism.
- `Security Attributes`: A principal has a set of security attributes associated with it. These security attributes are related to the authentication and authorization mechanisms. Examples are: security permissions, credentials for a principal.
- `Credential`: A credential contains or references security information that can authenticate a principal to additional services. A principal acquires a credential upon authentication or from another principal that allows its credential to be used (the latter is termed `principal delegation`).
- `End user:` An end user is an entity (human or service) that acts as a source of a request to an application. An end user is represented as a security principal within a `Subject` as specified in the JAAS framework [7].
- `Initiating Principal`: The security principal representing the end-user that interacts directly with the application. An end-user can authenticate using either a web client or an application client.
- `Caller Principal`: A principal that is associated with an application component instance during a method invocation. For example, an EJB instance can call `getCallerPrincipal` method to get the principal associated with the current security context.
- `Resource Principal`: A security principal under whose security context a connection to an EIS instance is established.
- `Security domain`: A scope within which certain common security mechanisms and policies are established. This specification does not specify the scope of a security domain. An enterprise can contain more than one security domain. Thus an application server and an EIS could either be in the same or different security domains. The Security Scenarios on page 159 provide illustrative examples of how security domains can be setup and managed.

In a managed environment, application components are deployed in web or EJB containers. When a method gets invoked on a component, the principal associated with the component instance is termed a caller principal.

The relationship between an initiating principal and a caller principal depends on the principal delegation option for inter-container and inter-component calls. This form of principal delegation is out of the scope of the connector architecture.

The relationship of a resource principal and its security attributes (example: credentials, access privileges) to an initiating / caller principal depends on how the resource principal has been setup by the system administrator or deployer.

Refer to section 8.2 for details on interfaces and classes that are used to represent a resource principal and its credentials.

## 7.4 Application Security Model

**Note**: The following section is a brief summary of the security model from the perspective of an application component provider. The reader should refer to the relevant specifications for more details.

The application component requests a connection to be established under the security context of a resource principal. The security context includes security attributes—access privileges, authorization level—for a resource principal. Once a connection is established successfully, all application-level invocations to the EIS instance using the connection happen under the security context of the resource principal.

The application component provider has the following two choices related to EIS sign-on:

- Allow the deployer to set up the resource principal and EIS sign-on information. For example, the deployer sets user name and password for establishing a connection to an EIS instance.
- Perform sign-on to an EIS from the component code by providing explicit security information for a resource principal.

The application component provider uses a deployment descriptor element (example: `res-auth` for EJB components) to indicate the requirements for one of the above two approaches. If the `res-auth` element is set to `Application`, the component code performs a programmatic sign-on to the EIS; if the `res-auth` element is `Container`, the application server takes the responsibility of setting up and managing EIS sign-on.

### 7.4.1 Scenario: Container-managed Sign-on

The application component provider sets the `res-auth` deployment descriptor element to be `Container` letting the application server take the responsibility of managing EIS sign-on.

The deployer sets up the principal mapping such that the user account for connecting to the EIS instance is always `eStoreUser`. The deployer also configures the authentication data (example, password) needed to authenticate `eStoreUser` to the EIS.

The component code invokes the `getConnection` method on the `ConnectionFactory` instance with no security-related parameters. The component relies on the application server to manage sign-on to the EIS instance based on the security information configured by the deployer.

```
// Method in an application component
Context initctx = new InitialContext();

// perform JNDI lookup to obtain connection factory
javax.resource.cci.ConnectionFactory cxf =
           (javax.resource.cci.ConnectionFactory)initctx.lookup(
                  “java:comp/env/eis/MyEIS”);

// Invoke factory to obtain a connection. The security
// information is not passed in the getConnection method
javax.resource.cci.Connection cx = cxf.getConnection();
...
```

### 7.4.2 Scenario: Component-Managed Sign-on

The application component provider sets the `res-auth` element to be `Application`.

The component code performs a programmatic sign-on to the EIS. The application component passes explicit security information (username, password) to the `getConnection` method of the `ConnectionFactory` instance.

```
// Method in an application component
Context initctx = new InitialContext();

// perform JNDI lookup to obtain connection factory
```

```
javax.resource.cci.ConnectionFactory cxf =
            (javax.resource.cci.ConnectionFactory)initctx.lookup(
                  “java:comp/env/eis/MyEIS”);

// Invoke factory to obtain a connection
com.myeis.ConnectionSpecImpl properties = //.. get a new ConnectionSpec
properties.setUserName(“...”);
properties.setPassword(“...”);
javax.resource.cci.Connection cx = cxf.getConnection(properties);
...
```

## 7.5 EIS Sign-on

Creating a new physical connection requires a sign-on to an EIS instance. Changing the security context on an existing physical connection can also require EIS sign-on; the latter is termed re-authentication.

An EIS sign-on typically involves one or more of the following steps. (This section explains all these mechanisms.):

- Determining a resource principal under whose security context a physical connection to an EIS will be established.
- Authentication of a resource principal if it is not already authenticated.
- Establishing a secure association between the application server and the EIS. This enables additional security mechanisms (example, data confidentiality and integrity) to be applied to communication between the two entities.
- Access control to EIS resources.

### 7.5.1 Authentication Mechanism

An application server and an EIS collaborate to ensure the proper authentication of a resource principal which establishes a connection to an underlying EIS. The connector architecture identifies the following as the commonly-supported authentication mechanisms:

- `BasicPassword`: Basic user-password- based authentication mechanism specific to an EIS
- `Kerbv5`: Kerberos version 5-based authentication mechanism

The `authentication-mechanism-type` element is used in the deployment descriptor to specify whether or not a resource adapter supports a specific authentication mechanism. [Refer to section 10.6 for more details on the specification of the deployment descriptor for a resource adapter.]

The connector architecture does not require that a specific authentication mechanism be supported by an application server and an EIS. An application server may support any other authentication mechanisms for EIS sign-on. The connector security architecture is independent of security mechanisms.

### 7.5.2 Resource Principal

When an application component requests a connection from a resource adapter, the connection request is made under the security context of a resource principal. The deployer can set a resource principal based on the following options:

- **Configured Identity:** In this case, a resource principal has its own configured identity and security attributes independent of the identity of the initiating/caller principal. The identity of resource principal can be configured either through a configuration of the principal at deployment time or specified dynamically by a component at the connection creation. The scenario EStore Application on page 159 illustrates an example where

connections to an EIS are always established under the security context of a valid EIS user account. This happens independent of the initiating or caller principal. For example: if a caller principal is `A`, then the configured resource principals can be `B` and `C` on two different EIS instances; where `A`, `B`, and `C` are independent identities.

- **Principal Mapping**: A resource principal is determined by mapping from the identity and/or security attributes of the initiating/caller principal. In this case, a resource principal does not inherit identity or security attributes of a principal that it has been mapped from; the resource principal gets its identity and security attributes based on the mapping. For example: if caller principal has identity `A`, then the mapped resource principal is `mapping(A,EIS1)` and `mapping(A, EIS2)` on two different EIS instances.
- **Caller Impersonation**: A resource principal acts on behalf of an initiating/caller principal. Acting on behalf of a caller principal requires that the caller's identity and credentials be delegated to the EIS. The mechanism by which this is accomplished is specific to a security mechanism and an application server implementation. An example of the impersonation is shown in the scenario Employee Self Service Application on page 161.

  In some scenarios, a caller principal can be a delegate of an initiating principal. In this case, a resource principal transitively impersonates an initiating principal.

  The support for principal delegation is typically specific to a security mechanism. For example, Kerberos supports a mechanism for the delegation of authentication. [Refer to Kerberos v5 specification for more details]. The security technology specific details are out of scope of the connector architecture.
- **Credentials Mapping**: This mechanism may be used when an application server and EIS support different authentication domains. For example, the initiating principal has been authenticated and has public key certificate-based credentials. The security environment for the EIS is configured with the Kerberos authentication service. The application server is configured to map the public key certificate-based credentials associated with the initiating principal to the Kerberos credentials. In this case, the resource principal is the same as the caller principal with the mapped credentials.

  In the case of credential mapping, the mapped resource principal has the same identity as the initiating/caller principal. For example, a principal with identity `A` has initial credentials `cred(A,mech1)` and has credentials `cred(A,mech2)` after mapping. The `mech1` and `mech2` represents different mechanism types.

### 7.5.3 Authorization Model

The authorization checking to ensure that a principal has access to an EIS resource can be applied at either (or both) of the following:

- At the EIS.
- At the application server.

Authorization checking at the target EIS can be done in an EIS-specific way and is not specified here. For example, an EIS can define its access control policy (in a security technology dependent) in terms of its specific security roles and permissions.

Authorization checking can also be done at the application server level. For example, an application server can allow a principal to create a connection to an EIS only if the principal is authorized to do so. J2EE containers (such as EJB and Servlet containers) support both programmatic and declarative security that can be used to define authorization policies. Programmatic and declarative security are defined in the individual specifications (refer to the EJB and Servlet specifications for more details). An application component developer developing components for EIS access must follow the requirements defined in these specifications.

### 7.5.4 Secure Association

The communication between an application server and an EIS can be subject to security threats (for example, data modification, loss of data). Establishing a secure association counters such threats. A secure association is a shared security information that allows a component on the application server to communicate securely with an EIS.

The establishment of a secure association can include several steps:

- The resource principal is authenticated to the EIS; this may require that the target principal in the EIS domain authenticate itself back to the application server. A target principal can be setup by the system administrator as a security principal associated with a running EIS instance or specific EIS resource.
- Negotiating a quality of protection, such as confidentiality or integrity.
- A pair of communicating entities—an application server and an EIS instance—establish a shared security context using the credentials of the resource principal. The security context encapsulates shared state information, required so that communication between the application server and the EIS can be protected through integrity and confidentiality mechanisms. Examples of shared state information that is part of a security context are cryptographic keys and message sequence numbers.

A secure association between an application server and an EIS is always established by the resource adapter implementation. Note that a resource adapter library runs within the address space of the application server.

A resource adapter can use any security mechanism to establish the secure association. GSS-API (refer to IETF draft on GSS-API v2[5]) is an example of such a mechanism. Note that the connector architecture does not require use of the GSS-API by a resource adapter or application server.

The configuration of a mechanism for establishing secure association is outside the scope of the connector architecture. This includes setting up the desired quality of protection during secure communication.

Once a secure association is established successfully, the connection is associated with the security context of the resource principal. Subsequently, all application-level invocations to the EIS instance using the connection happen under the security context of the resource principal.

## 7.6 Roles and Responsibilities

This section describes various roles involved in the security architecture. It also describes responsibilities of each role from the security perspective.

The roles and responsibilities of the application component provider and deployer are specified in detail in the respective J2EE component model specifications.

### 7.6.1 Application Component Provider

The following features are common across different J2EE component models from the perspective of an application component provider:

- An application component provider invariably avoids the burden of securing its application and focuses on developing the business functionality of its application.
- A security-aware application component provider can use a simple programmatic interface to manage security at an application level. The programmatic interface enables an application component provider to program access control decisions based on the security context—principal, role—associated with the caller of a method and to manage programmatic sign-on to an EIS.

- An application component provider specifies security requirements for its application declaratively in a deployment descriptor. The security requirements include security roles, method permissions, and an authentication approach for EIS sign-on.
- More qualified roles—application server vendor, deployer, system administrator—have the responsibility of satisfying overall security requirements (through the deployment mechanism for resource adapters and components) and managing the security environment.

### 7.6.2 Deployer

The deployer specifies security policies that ensure secure access to the underlying EISs from application components. The deployer adapts the *intended* security view of an application for EIS access, specified through a deployment descriptor, to the *actual* security mechanisms and policies used by the application server and EISs in the target operational environment. The deployer uses tools to accomplish the above task.

The output of the deployer's work is a security policy descriptor specific to the operational environment. The format of the security policy descriptor is specific to an application server.

The deployer performs the following deployment tasks for each connection factory reference declared in the deployment descriptor of an application component:

- Provides a connection factory specific security configuration that is needed for opening and managing connections to an EIS instance.
- Binds the connection factory reference in the deployment descriptor of an application component to the JNDI registered reference for the connection factory. Refer to section 10.5 for the JNDI configuration of a connection factory during deployment of a resource adapter. The deployer can use the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the connection factory.
- If the value of the `res-auth` deployment descriptor element is `Container`, the deployer is responsible for configuring the security information for EIS sign-on. For example, the deployer sets up the principal mapping for EIS sign-on.

### 7.6.3 Application Server

The application server provides a security environment with specific security policies and mechanisms that support the security requirements of the deployed application components and resource adapters, thereby ensuring a secure access to the connected EISs.

The typical responsibilities of an application server are as follows:

- Provide tools to set up security information for a resource principal and EIS sign-on when `res-auth` element is set to `Container`. This includes support for principal delegation and mapping for configuring a resource principal.
- Provide tools to support management and administration of its security domain. For example, security domain administration can include setting up and maintaining both underlying authentication services and trusts between domains, plus managing principals (including identities, keys, attributes). Such administration is typically security technology specific and is outside the scope of the connector architecture.
- Support a single sign-on mechanism that spans the application server and multiple EISs. The security mechanisms and policies through which single sign-on is achieved are outside the scope of the connector architecture.

The Appendix C specifies how JAAS can be used by an application server to support the requirements of the connector security architecture.

### 7.6.4 EIS Vendor

The EIS provides a security infrastructure and environment that supports the security requirements of the client applications. An EIS can have its own security domain with a specific set of

security policies and mechanisms or it can be set up as part of an enterprise-wide security domain.

### 7.6.5 Resource Adapter Provider

The resource adapter provider provides a resource adapter that supports the security requirements of the underlying EIS.

The resource adapter implements the security contract specified as part of the connector architecture. Chapter 8 specifies the security contract and related requirements for a resource adapter.

The resource adapter specifies its security capabilities and requirements through its deployment descriptor. Section 10.6 specifies a standard deployment descriptor for a resource adapter.

### 7.6.6 System Administrator

The system administrator typically works in close association with administrators of multiple EISs that have been deployed in an operational environment. The system administration tasks can also be performed by the deployer.

The following tasks are illustrative examples of the responsibilities of the system administrator:

- Setup an operational environment based on the technology and requirements of the authentication service, and if an enterprise directory is supported.
- Configure the user account information for both the application server and the EIS in the enterprise dierectory.The user account information from the enterprise directory can then be used for authentication of users requesting connectivity to the EIS.
- Establish a password synchronization mechanism between the application server and the EIS. This ensures that the user's security information is identical on the application server and the EIS. When an EIS requires authentication, the application server passes the user's password to the EIS.

# 8 Security Contract

This chapter specifies the security contract between the application server and the EIS. It also specifies the responsibilities of the resource adapter provider and the application server vendor for supporting the security contract.

This chapter references the following chapters and documents:

- The security model specified in the J2EE platform specification [8].
- Security architecture specified in Chapter 7.
- Security scenarios based on the connector architecture [refer to Appendix: Security Scenarios on page 159].

## 8.1 Security Contract

The security contract between the application server and the resource adapter extends the connection management contract (described in Chapter 5) by adding security specific details.

This security contract supports EIS sign-on by:

- Passing the connection request from the resource adapter to the application server, enabling the latter to hook-in security services.
- Propagation of the security context —JAAS `Subject` with principal and credentials—from the application server to the resource adapter.

## 8.2 Interfaces/Classes

The security contract includes the following classes and interfaces:

### 8.2.1 Subject

The following text has been used from the JAAS specification. For a detailed specification, refer to JAAS documents:

A `Subject` represents a grouping of related information for a single entity, such as a person. Such information includes the Subject's identities and its security-related attributes (for example, passwords and cryptographic keys). A `Subject` can have multiple identities. Each identity is represented as a `Principal` within the `Subject`. A `Principal` simply binds a name to a `Subject`.

A `Subject` can also own security-related attributes, which are referred to as `Credentials`. Sensitive credentials that require special protection, such as private cryptographic keys, are stored within a private credential set.

The `Credentials` intended to be shared, such as public key certificates or Kerberos server tickets, are stored within a public credential set. Different permissions are required to access and modify different credential sets.

The `getPrincipals` method retrieves all the principals associated with a `Subject`. The methods `getPublicCredentials` and `getPrivateCredentials` respectively retrieve all the public or private credentials belonging to a `Subject`. The methods defined in the `Set` class modify the returned set of principals and credentials.

### 8.2.2 ResourcePrincipal

The interface `java.security.Principal` represents a resource principal. The following code extract shows the `Principal` interface:

```
public interface java.security.Principal {
     public boolean equals(Object another);
     public String getName();
     public String toString();
     public int hashCode();
}
```

The method `getName` returns the name of a resource principal.

An application server should use the `Principal` interface (or any derived interface) to pass a resource principal as part of a `Subject` to a resource adapter.

### 8.2.3 GenericCredential

The interface `javax.resource.spi.security.GenericCredential` defines a security mechanism independent interface for accessing the security credential of a resource principal.

The `GenericCredential` interface provides a Java wrapper over an underlying mechanism specific representation of a security credential. For example, the `GenericCredential` interface can be used to wrap Kerberos credentials.

The connector architecture does not define any standard format and requirements for security mechanism specific credentials. For example, a security credential wrapped by a `GenericCredential` interface can have a native representation specific to an operating system.

> **Note**: A contract for the representation of mechanism-specific credentials must be established between an application server and a resource adapter outside the scope of the connector architecture. This includes requirements for the exchange of mechanism-specific credentials between a JAAS module and GSS provider. Refer to Appendix C: JAAS based Security Architecture for details on JAAS-based security architecture.

The `GenericCredential` interface enables a resource adapter to extract information about a security credential. The resource adapter can then manage an EIS sign-on for a resource principal by either:

- Using the credentials in an EIS specific manner if the underlying EIS supports the security mechanism type represented by the `GenericCredential` instance, or,
- Using GSS-API [5] if the resource adapter and underlying EIS instance support GSS-API.

#### Interface

The following code extract shows the `GenericCredential` interface:

```
public interface javax.resource.spi.security.GenericCredential {
     public String getName();
     public String getMechType();
     public byte[] getCredentialData()
            throws javax.resource.spi.SecurityException;

     public boolean equals(Object another);
     public int hashCode();

}
```

The GenericCredential interface supports a set of getter methods to obtain information about a security credential.

The method getName returns the name of the resource principal associated with a GenericCredential instance.

The method getMechType returns the mechanism type for the GenericCredential instance. The mechanism type definition for GenericCredential must be consistent with the Object Identifier (OID) based representation specified in the GSS [5] specification. In the GenericCredential interface, the mechanism type is returned as a stringified representation of the OID specification.

The GenericCredential interface can be used to get security data for a specific security mechanism. An example is authentication data required for establishing a secure association with an EIS instance on behalf of the associated resource principal. The getCredentialData method returns the credential representation as an array of bytes. Note that the connector architecture does not define a standard format for the returned credential data.

**Implementation**

If an application server supports deployment of a resource adapter which supports GenericCredential as part of the security contract, then the application server is required to provide an implementation of the GenericCredential interface. Refer to the deployment descriptor specification in Section 10.6 for details on how a resource adapter specifies its support for GenericCredential.

### 8.2.4 PasswordCredential

The class javax.resource.spi.security.PasswordCredential acts as a holder of username and password. This class enables an application server to pass username and password to the resource adapter through the security contract.

The method getUserName on PasswordCredential class gets the name of the resource principal. The interface java.security.Principal represents a resource principal.

The PasswordCredential class is required to implement equals and hashCode method.

```
public final class javax.resource.spi.security.PasswordCredential
            implements java.io.Serializable {
      public PasswordCredential(String userName, char[] password) { ... }
      public String getUserName() { ... }
      public char[] getPassword() { ... }

      public ManagedConnectionFactory getManagedConnectionFactory()
                  { ... }
      public void setManagedConnectionFactory(
                  ManagedConnectionFactory mcf) { ... }

      public boolean equals(Object other) { ... }
      public int hashCode() { ... }
}
```

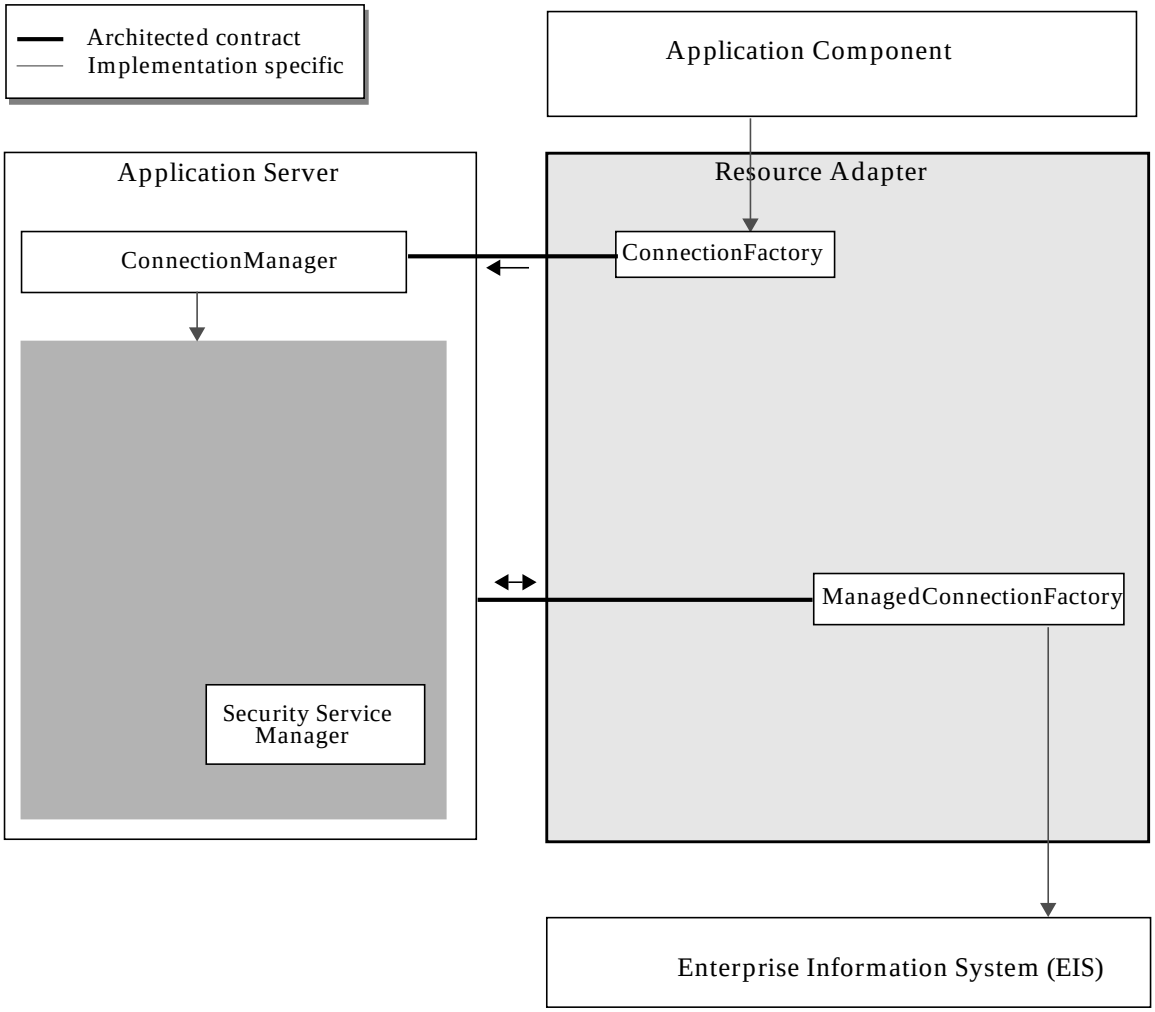
The method getManagedConnectionFactory returns the ManagedConnectionFactory instance for which the user name and password has been set by the application server. Refer to the contract for ManagedConnectionFactory to see how a resource adapter uses this method.

### 8.2.5 ConnectionManager

The method ConnectionManager.allocateConnection is called by the resource adapter's connection factory instance. This method lets the resource adapter pass a connection request to the application server, so that the latter can hook-in security and other services.

```
public interface javax.resource.spi.ConnectionManager
     extends java.io.Serializable {

     public Object allocateConnection(
           ManagedConnectionFactory mcf,
           ConnectionRequestInfo cxRequestInfo)
                 throws ResourceException;
}
```

**FIGURE 27.0** Security Contract




Depending on whether application server or application component is configured to be responsible for managing EIS sign-on (refer to Section 7.6.1), the resource adapter calls the `ConnectionManager.allocateConnection` method in one of the following ways:

- **Option—Container Managed Sign-on**: The application component passes no security information in the `getConnection` method and the application server is configured to manage EIS sign-on.

The application server provides the required security information for the resource principal through its configured security policies and mechanisms (for example, principal mapping). The application server requests the authentication of the resource principal to the EIS either itself or passes authentication responsibility to the resource adapter. This aspect is explained later in the specification of the `ManagedConnectionFactory` interface.

- **Option—Component Managed Sign-on**: In this case, the application component provides explicit security information in the `getConnection` method. The resource adapter invokes the `allocateConnection` method by passing security information in the `ConnectionRequestInfo` parameter. Since the security information in the `ConnectionRequestInfo` is opaque to the application server, the application server should rely on the resource adapter to manage EIS sign-on (explained in the `ManagedConnectionFactory` interface specification under option C).

### 8.2.6 ManagedConnectionFactory

The following code extract shows the methods on the `ManagedConnectionFactory` interface that are relevant to the security contract:

```
public interface javax.resource.spi.ManagedConnectionFactory
     extends java.io.Serializable {

     public ManagedConnection createManagedConnection(
            javax.security.auth.Subject subject,
            ConnectionRequestInfo cxRequestInfo)
                   throws ResourceException;
     ...
}
```

During the JNDI lookup, the `ManagedConnectionFactory` instance is configured by the application server with a set of configuration properties. These properties include default security information and EIS instance specific information (hostname, port number) required for initiating a sign-on to the underlying EIS during the creation of a new physical connection.

The default security configuration on a `ManagedConnectionFactory` can be overridden by security information provided either by a component (in component managed sign-on) or by container (in container managed sign-on).

The method `createManagedConnection` is used by the application server when it requests resource adapter to create a new physical connection to the underlying EIS.

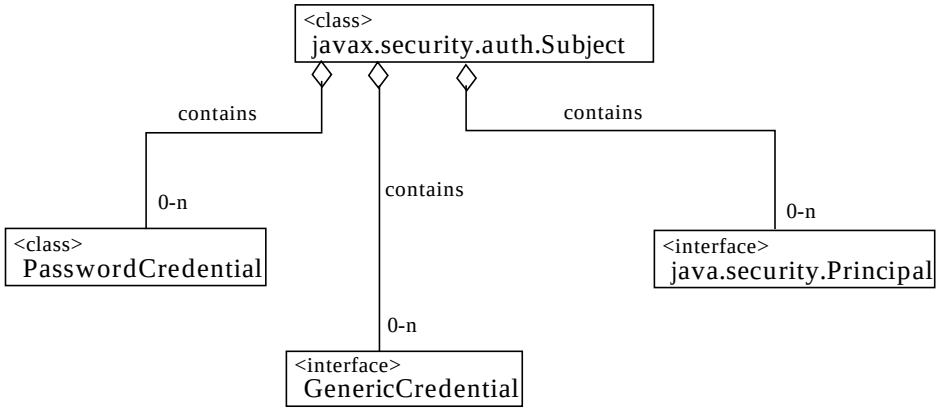
#### Contract for Application Server

The application server may provide specific security services (principal mapping and delegation, single sign-on) before using the security contract with the resource adapter. For example, the application server can map the caller principal to a resource principal before calling the method `createManagedConnection` to create a new connection (under the security context of the resource principal).

In the container-managed sign-on, the application server is responsible for creating a `Subject` instance using its implementation-specific security mechanisms and configuration. The creation should happen before the application server calls the method `createManagedConnection` on the `ManagedConnectionFactory`. Resource adapter is driven by application server and acts as consumer of security information in the created `Subject`.

If the application server maintains a cache of the security credentials (example, Kerberos TGT), then the application server should reuse the credentials as part of the newly created `Subject` instance. For example, the application server uses the method `Subject.getPrivateCredentials().add(credential)` to add a credential to the private credential set.

**FIGURE 28.0** Security Contract: Subject Interface and its Containment Hierarchy




The above diagram shows the relationship between `Subject`, `Principal`, `PasswordCredential` and `GenericCredential` interfaces. Note that in the following options A and B defined for `createManagedConnection` method invocation, the `Subject` instance contains a single resource principal (represented as `java.security.Principal`) and multiple credentials.

The application server is required to use one of the following options for invoking the method `createManagedConnection`:

- **Option A**: The application server invokes the method `createManagedConnection` by passing in a non-null `Subject` instance that carries a **single** resource principal and its corresponding password-based credentials (represented by the class `PasswordCredential` that provides the user name and password). The `PasswordCredential` should be set in the `Subject` instance as a part of the private credential set. Note that the passed `Subject` can contain multiple `PasswordCredential` instances.

  The resource adapter extracts the user name and password from this `Subject` instance (by looking for `PasswordCredential` instance in the `Subject`) and uses this security information to sign-on to the EIS instance during the connection creation.

- **Option B**: The application server invokes the method `createManagedConnection` method by passing in a non-null `Subject` instance that carries a **single** resource principal and its security credentials. In this option, credentials are represented through the `GenericCredential` interface. A typical example is a `Subject` instance with Kerberos credentials.

  For example, an application server may use this option for `createManagedConnection` method invocation when the resource principal is impersonating the caller/initiating principal and has valid credentials acquired through impersonation. An application server may also use this option for principal mapping scenarios with credentials of a resource principal represented through the `GenericCredential` interface.

  Note that sensitive credentials requiring special protection, such as private cryptographic keys, are stored within a private credential set, while credentials intended to be shared, such as public key certificates or Kerberos server tickets, are stored within a public credential set. The two methods `getPrivateCredentials` and `getPublicCredentials` should be used accordingly.

  In case of Kerberos mechanism type, the application server must pass the principal's TGT (ticket granting ticket) to a resource adapter in a private credential set.

The resource adapter uses the resource principal and its credentials from the `Subject` instance to go through the EIS sign-on process before creating a new connection to the EIS.

- **Option C**: The application server invokes method `createManagedConnection` by passing a `null Subject` instance. The application server is required to use this option for the component-managed sign-on case. In this option, security information is carried in the `ConnectionRequestInfo` instance. The application server does not provide any security information that can be used by the resource adapter for managing EIS sign-on.

During the deployment of a resource adapter, the application server must be configured to use one of the above specified invocation options. Refer the deployment chapter 10 for more details.

### Contract for Resource Adapter

A resource adapter can do EIS sign-on and connection creation in an implementation-specific way or it can use the `GSS-API`. The latter option is specified in the appendix on page 166. A resource adapter has the following options (corresponding to the options for an application server) for handling the invocation of the method `createManagedConnection`:

- **Option A**: The resource adapter explicitly checks whether the passed `Subject` instance carries a `PasswordCredential` instance using the `Subject.getPrivateCredentials` method.

  Note that the security contract assumes that a resource adapter has the necessary security permissions to extract a private credential set from a `Subject` instance. The specific mechanism through which such permission is set up is outside the scope of the connector architecture.

  If the `Subject` instance contains a `PasswordCredential` instance, the resource adapter extracts the user name and password from the `PasswordCredential`. It uses the security information to authenticate the resource principal (corresponding to the user name) to the EIS during the creation of a connection. In this case, the resource adapter uses an authentication mechanism that is EIS specific.

  Since a `Subject` instance can carry multiple `PasswordCredential` instances, a `ManagedConnectionFactory` should only use a `PasswordCredential` instance that has been specifically passed to it through the security contract. The method `getManagedConnectionFactory` enables a `ManagedConnectionFactory` instance to determine whether or not a `PasswordCredential` instance is to be used for sign-on to the target EIS instance. The `ManagedConnectionFactory` implementation uses the `equals` method to compare itself with the passed instance.

- **Option B**: The resource adapter explicitly checks whether passed `Subject` instance carries a `GenericCredential` instance using the methods `getPrivateCredentials` and `getPublicCredentials` defined on the `Subject` interface.

  In case of Kerberos mechanism type, the resource adapter must extract Kerberos credentials using the method `getPrivateCredentials` on the `Subject` interface.

  The resource adapter uses the resource principal and its credentials (represented by the `GenericCredential` interface) in the `Subject` instance to go through the EIS sign-on process. For example, this option is used for Kerberos-based credentials that have been acquired by the resource principal through impersonation.

  A resource adapter uses the getter methods defined on the `GenericCredential` interface to extract information about the credential and its principal. If a resource adapter is using GSS mechanism, the resource adapter uses a reference to the `GenericCredential` instance in an opaque manner and is not required to understand any mechanism-specific credential representation. However, a resource adapter may need to interpret credential representation if the resource adapter initiates authentication in an implementation-specific manner.

- **Option C**: If the application server invokes `ManagedConnectionFactory.createManagedConnection` with a `null Subject` instance, then a resource adapter has the following options:
  - Resource adapter should extract security information passed through the `ConnectionRequestInfo` instance. The resource adapter should authenticate resource principal by combining the configured security information on the `ManagedConnectionFactory` instance with the security information passed through the `ConnectionRequestInfo` instance. The default for resource adapter is to allow the security information in the `ConnectionRequestInfo` parameter to override the configured security information in the `ManagedConnectionFactory` instance.
  - If the resource adapter does not find any security configuration in the `ConnectionRequestInfo`, resource adapter uses the default security configuration on the `ManagedConnectionFactory` instance.

### 8.2.7 ManagedConnection

A resource adapter can re-authenticate a physical connection (one that already exists in the connection pool under a different security context) to the underlying EIS. A resource adapter does re-authentication when an application server calls `getConnection` method with a security context (passed as a `Subject` instance) different from the context previously associated with the physical connection.

If a resource adapter supports re-authentication, the `matchManagedConnections` method (on `ManagedConnectionFactory`) may return a matched `ManagedConnection` instance with the assumption that `ManagedConnection.getConnection` method will later switch the security context through re-authentication. Note that `matchManagedConnections` method should consider a `ManagedConnection` instance as immutable; there is no authentication involved in the `matchManagedConnections` method.

Support for re-authentication depends on whether an underlying EIS supports re-authentication mechanism for existing physical connections. If a resource adapter does not support re-authentication, the `getConnection` method should throw `javax.resource.spi.SecurityException` if the passed `Subject` in the `getConnection` method is different from the security context associated with the `ManagedConnection` instance.

```
public interface javax.resource.spi.ManagedConnection {
     public Object getConnection(
                  javax.security.auth.Subject subject,
                  ConnectionRequestInfo cxRequestInfo)
                                  throws ResourceException;
     ...
}
```

The `getConnection` method returns a new connection handle. If re-authentication is successful, the resource adapter has changed the security context of the underlying `ManagedConnection` instance to that associated with the passed `Subject` instance.

A resource adapter has the following options for handling `ManagedConnection.getConnection` invocation if it supports re-authentication:

- **Option A**: The resource adapter extracts `PasswordCredential` instance from the `Subject` and performs an EIS-specific authentication. This option is similar to option A defined in the specification of the method `createManagedConnection` on the interface `ManagedConnectionFactory`.

- **Option B**: The resource adapter extracts `GenericCredential` instance from the `Subject` and manages authentication either through the GSS mechanism or an implementation-specific mechanism. This option is similar to option B defined in the specification of the method `createManagedConnection` on the interface `ManagedConnectionFactory`.
- **Option C**: In this case, the `Subject` parameter is `null`. The resource adapter extracts security information from the `ConnectionRequestInfo` (if there is any) and performs authentication in an implementation-specific manner. This option is similar to option C defined in the specification of the method `createManagedConnection` on the interface `ManagedConnectionFactory`.

## 8.3 Requirements

The following are the requirements defined by the security contract:

### Resource Adapter

The following are the requirements defined for a resource adapter:

- Resource adapter is required to support the security contract by implementing the method `ManagedConnectionFactory.createManagedConnection`.
- Resource adapter is not required to support re-authentication as part of its `ManagedConnection.getConnection` method implementation.
- Resource adapter is required to specify its support for the security contract as part of its deployment descriptor. The relevant deployment descriptor elements are [refer section 10.6 for a detailed specification]: `authentication-mechanism`, `authentication-mechanism-type`, `reauthentication-support` and `credential-interface`.

### Application Server

The following are the requirements defined for an application server:

- Application server is required to use the method `ManagedConnectionFactory.createManagedConnection` to pass the security context to the resource adapter during EIS sign-on.
- Application server is required to be capable of using options - A and C - as specified in the section 8.2.6 for the security contract.
- Application server provides an implementation of `GenericCredential` interface if the following conditions are both true:
    - Application server supports authentication mechanisms (specified as `authentication-mechanism-type` in the deployment descriptor) other than `BasicPassword` mechanism. For example, application server should implement `GenericCredential` interface to support `kerbv5` authentication mechanism type.
    - Application server supports deployment of resource adapters that are capable of handling `GenericCredential` (and thereby option B as specified in section 8.2.6) as part of the security contract.
- Application server is required to implement the method `allocateConnection` in its `ConnectionManager` implementation.
- Application server is required to configure its use of the security contract based on the security requirements specified by the resource adapter in its deployment descriptor. For example, if a resource adapter specifies that it supports only `BasicPassword` authentication, application server should use the security contract to pass `PasswordCredential` instance to the resource adapter.

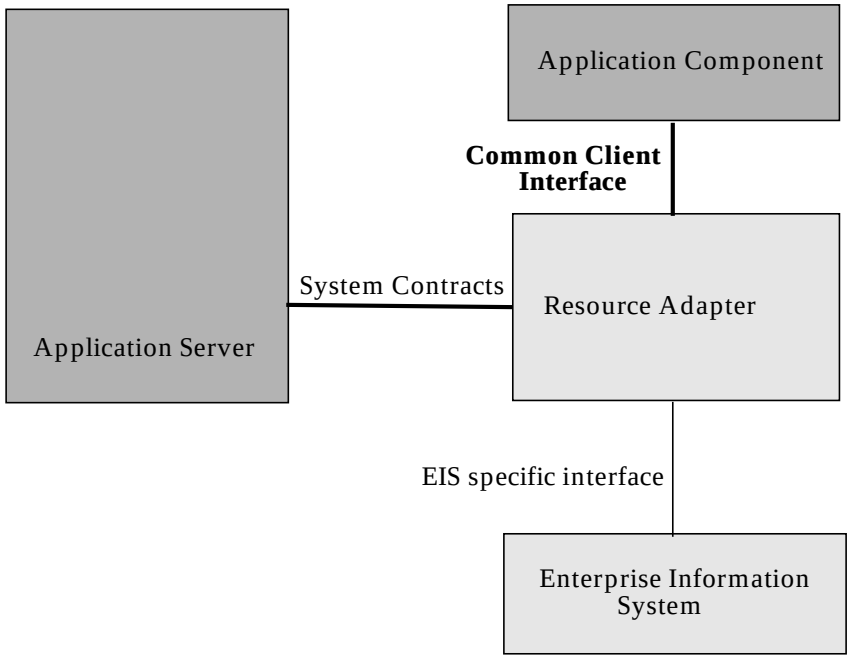
# 9 Common Client Interface

The following chapter specifies the Common Client Interface (CCI).

## 9.1 Overview

The CCI defines a standard client API for application components. The CCI enables application components and Enterprise Application Integration (EAI) frameworks to drive interactions across heterogeneous EISs using a common client API. Figure 29.0 shows a high-level view of the CCI and its relationship to other application components.

**FIGURE 29.0** Common Client Interface




## 9.2 Goals

The CCI is designed with the following goals:

- It defines a *remote function-call* interface that focuses on executing functions on an EIS and retrieving the results. The CCI can form a base level API for EIS access on which higher level functionality can be built.
- It is targeted primarily towards application development tools and EAI frameworks.
- Although it is simple, it has sufficient functionality and an extensible application programming model.

- It provides an API that both leverages and is consistent with various facilities defined by the Java J2SE and J2EE platforms.
- It is independent of a specific EIS; for example: data types specific to an EIS. However, the CCI can be capable of being *driven* by EIS-specific metadata from a repository.

An important goal for the CCI is to complement existing standard JDBC API and not to replace this API. The CCI defines a common client API that is parallel to the JDBC for EISs that are not relational databases.

Since the CCI is targeted primarily towards application development tools and EAI vendors, it is not intended to discourage the use of JDBC APIs by these vendors. For example, an EAI vendor will typically combine JDBC with CCI by using the JDBC API to access relational databases and using CCI to access other EISs.

## 9.3 Scenarios

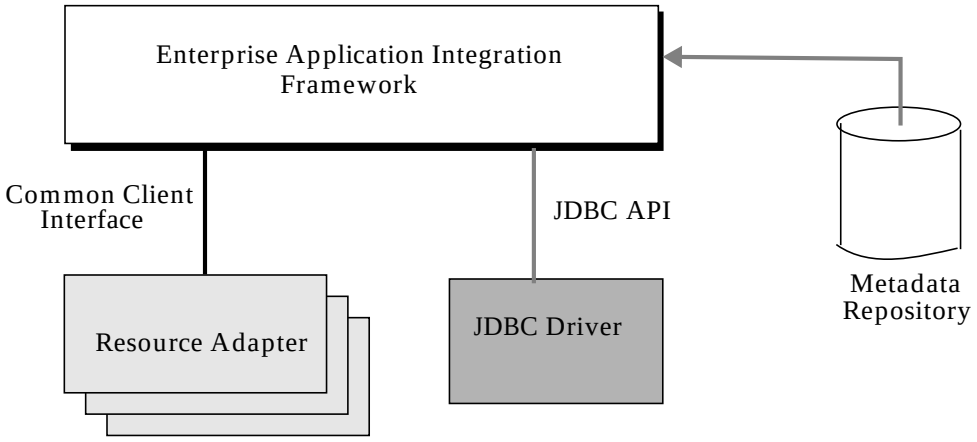
The following scenarios illustrate the use of CCI by enterprise tools and Enterprise Application Integration (EAI) vendors:

### 9.3.1 Enterprise Application Integration Framework

The EAI vendor uses the Common Client Interface as a standard way to plug-in resource adapters for heterogeneous EISs. The vendor provides an application integration framework on top of the functionality provided by the resource adapters. The framework uses the standard CCI interfaces to drive interactions with the connected EISs.

Figure 30.0 also shows the use of JDBC by the EAI framework for connecting to and accessing relational databases.

**FIGURE 30.0** Scenario: EAI Framework




### 9.3.2 Metadata Repository and API

An EAI or application development tool uses a metadata repository to drive CCI-based interactions with heterogeneous EISs. See Figure 30.0 and Figure 31.0 for illustrative examples. A repository may maintain meta information about functions (with type mapping information and data structures for the invocation parameters) existing on an EIS system.

**Note**: The specification of a standard repository API and metadata format is outside the scope of the current version 1.0 of the connector architecture.

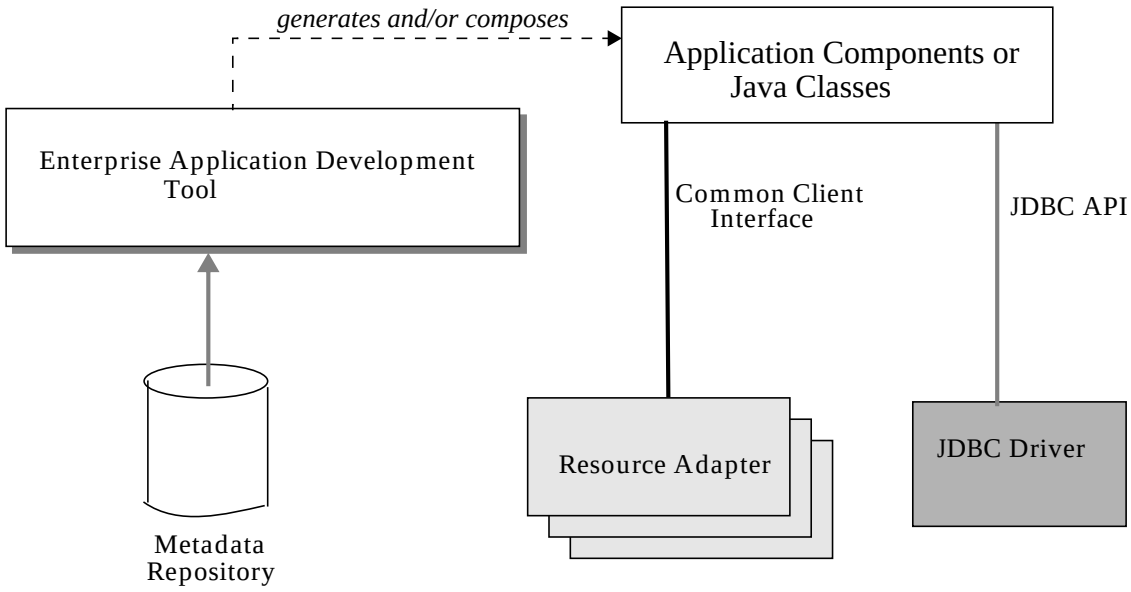
### 9.3.3 Enterprise Application Development Tool

The CCI functions as a plug-in contract for an application development tool that develops additional functionality around a resource adapter.

The application development tool generates Java classes based on the meta information accessed from a metadata repository. These Java classes encapsulate CCI-based interactions and expose a simple application programming model (typically based on the JavaBeans framework) to the application developers. An application component uses the generated Java classes for EIS access.

An application development tool can also compose or generate an application component that uses the generated Java classes for EIS access. See Figure 31.0.

**FIGURE 31.0** Scenario: Enterprise Application Development Tool



## 9.4 Common Client Interface

The CCI is divided in to the following parts:

- Connection-related interfaces that represent a connection factory and an application level connection:
  - `javax.resource.cci.ConnectionFactory`
  - `javax.resource.cci.Connection`
  - `javax.resource.cci.ConnectionSpec`
  - `javax.resource.cci.LocalTransaction`
- Interaction-related interfaces that enable a component to drive an interaction (specified through an `InteractionSpec`) with an EIS instance:
  - `javax.resource.cci.Interaction`
  - `javax.resource.cci.InteractionSpec`
- Data representation-related interfaces that are used to represent data structures involved in an interaction with an EIS instance:
  - `javax.resource.cci.Record`, `javax.resource.cci.MappedRecord` and `javax.resource.cci.IndexedRecord`
  - `javax.resource.cci.RecordFactory`
  - `javax.resource.cci.Streamable`
  - `javax.resource.cci.ResultSet`
  - `java.sql.ResultSetMetaData`
- Metadata related-interfaces that provide basic meta information about a resource adapter implementation and an EIS connection:
  - `javax.resource.cci.ConnectionMetaData`
  - `javax.resource.cci.ResourceAdapterMetaData`
  - `javax.resource.cci.ResultSetInfo`
- Additional classes: `javax.resource.ResourceException` and `javax.resource.cci.ResourceWarning`

See Figure 32.0 on page 102.

### 9.4.1 Requirements

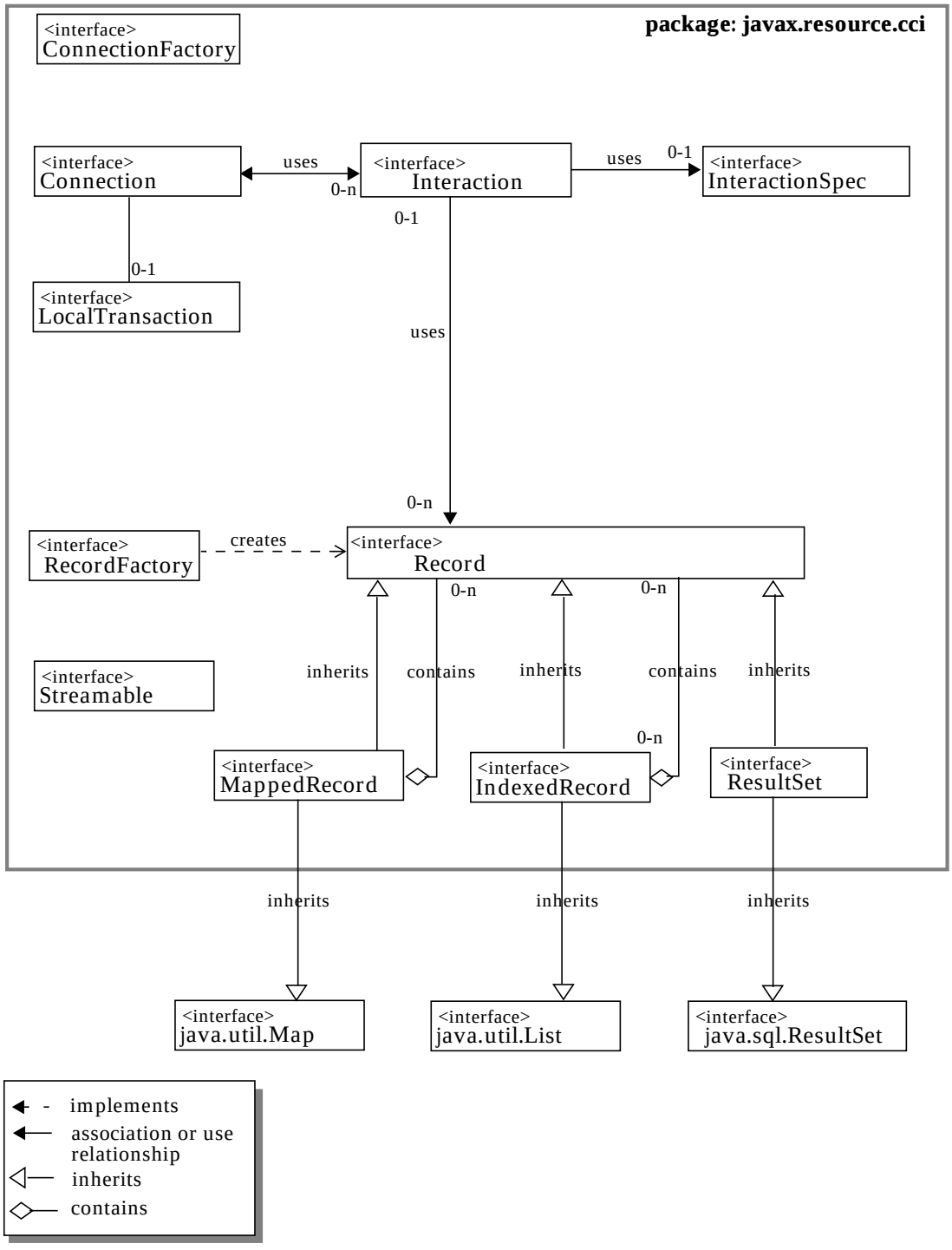
A resource adapter provider provides an implementation of the CCI interfaces as part of its resource adapter implementation. The connector architecture does not mandate that a resource adapter support the CCI interfaces as its client API.

> **Important**: A resource adapter is allowed to support a client API specific to its underlying EIS. An example of an EIS-specific client APIs is JDBC API for relational databases.

The connector architecture also allows a third party vendor to provide an implementation of CCI interfaces above a resource adapter. For example, a *base* resource adapter supports the system contracts and provides an EIS specific client API. A third party tools vendor may provide the CCI implementation above this *base* resource adapter.

The connector architecture also allows a resource adapter implementation to support all interfaces except the data representation-related interfaces. In this case, a third party vendor provides both the development-time and run-time aspects of data structures required to drive interactions with an EIS instance. The section on the `Record` interface specification describes this case in more detail.

**FIGURE 32.0** Class Diagram: Common Client Interface



## 9.5 Connection Interfaces

The following section specifies interfaces for the connection factory and application level connection.

### 9.5.1 ConnectionFactory

The `javax.resource.cci.ConnectionFactory` provides an interface for getting connection to an EIS instance. A component looks up a `ConnectionFactory` instance from the JNDI namespace and then uses it to get a connection to the EIS instance.

The following code extract shows the `ConnectionFactory` interface:

```
public interface javax.resource.cci.ConnectionFactory
            extends java.io.Serializable, javax.resource.Referenceable {

      public RecordFactory getRecordFactory()
                  throws ResourceException;

      public Connection getConnection()
                  throws ResourceException;
      public Connection getConnection(
            javax.resource.cci.ConnectionSpec properties)
                  throws ResourceException;

      public ResourceAdapterMetaData getMetaData()
                  throws ResourceException;

}
```

The `getConnection` method gets a connection to an EIS instance. The `getConnection` variant with no parameters is used when a component requires the container to manage EIS sign-on. In this case of the container-managed sign-on, the component does not pass any security information.

A component may also use the `getConnection` variant with `javax.resource.cci.ConnectionSpec` parameter, if it needs to pass any resource adapter specific security information and connection parameters. In the component-managed sign-on case, an application component passes security information (example: username, password) through the `ConnectionSpec` instance.

It is important to note that the properties passed through the `getConnection` method should be client-specific (example: user name, password, language) and not related to the configuration of a target EIS instance (example: port number, server name). The `ManagedConnectionFactory` instance is configured with complete set of properties required for the creation of a connection to an EIS instance. Configured properties on a `ManagedConnectionFactory` can be overridden by client-specific properties passed by an application component through the `getConnection` method. Refer section 10.4.1 for configuration of a `ManagedConnectionFactory`.

Note that in a managed environment, the `getConnection` method with no parameters is the recommended model for getting a connection. The container manages the EIS sign-on in this case.

The `ConnectionFactory` interface also provides a method to get a `RecordFactory` instance. The `ConnectionFactory` implementation class may throw a `javax.resource.NotSupportedException` from the method `getRecordFactory`.

**Requirements**

An implementation class for `ConnectionFactory` is required to implement the `java.io.Serializable` interface to support JNDI registration. A `ConnectionFactory` implementation class is also required to implement `javax.resource.Referenceable`. Note that the `javax.resource.Referenceable` interface extends the `javax.naming.Referenceable` interface. Refer section 10.5 for more details on JNDI based requirements for the `ConnectionFactory` implementation.

An implementation class for `ConnectionFactory` is required to provide a default constructor.

### 9.5.2 ConnectionSpec

The interface `javax.resource.cci.ConnectionSpec` is used by an application component to pass connection request-specific properties to the `getConnection` method.

The `ConnectionSpec` interface has been introduced to increase the toolability of the CCI. It is recommended that the `ConnectionSpec` interface be implemented as a JavaBean to support tools. The properties on the `ConnectionSpec` implementation class must be defined through the getter and setter methods pattern.

The following code extract shows the `ConnectionSpec` interface.

```
public interface javax.resource.cci.ConnectionSpec {
}
```

The CCI specification defines a set of standard properties for an `ConnectionSpec`. The properties are defined either on a derived interface or an implementation class of an empty `ConnectionSpec` interface. In addition, a resource adapter may define additional properties specific to its underlying EIS.

The following standard properties are defined by the CCI specification for `ConnectionSpec`:

- `UserName` name of the user establishing a connection to an EIS instance
- `Password` password for the user establishing a connection

An important point to note is about the relationship between `ConnectionSpec` and `ConnectionRequestInfo`. The `ConnectionSpec` is used at application level and is defined under the scope of CCI; while `ConnectionRequestInfo` is defined as part of the system contracts. Separate interfaces have been defined for these two to ensure the separation between CCI interfaces and system contracts; `ConnectionRequestInfo` has no explicit dependency on CCI. Note that in the 1.0 scope, a resource adapter may not implement CCI while it is required to implement system contracts. The mapping between CCI's `ConnectionSpec` and `ConnectionRequestInfo` is achieved in an implementation specific manner by a resource adapter.

### 9.5.3 Connection

A `javax.resource.cci.Connection` represents an application level connection handle that is used by a component to access an EIS instance. The actual physical connection associated with a `Connection` instance is represented by a `ManagedConnection`.

A component gets a `Connection` instance by using the `getConnection` method on a `ConnectionFactory` instance. A `Connection` instance may be associated with zero or more `Interaction` instances.

The following code extract shows the `Connection` interface:

```
public interface javax.resource.cci.Connection {
     public Interaction createInteraction() throws ResourceException;
```

```
        public ConnectionMetaData getMetaData() throws ResourceException;
        public ResultSetInfo getResultSetInfo() throws ResourceException;

        public LocalTransaction getLocalTransaction()
                                        throws ResourceException;

        public void close() throws ResourceException;
    }
```

The method `createInteraction` creates an `Interaction` instance associated with the `Connection` instance. An `Interaction` enables a component to access EIS data and functions.

The method `getMetaData` returns information about the EIS instance associated with a `Connection` instance. The EIS instance-specific information is represented by the `ConnectionMetaData` interface.

The method `getResultSetInfo` returns information on the result set functionality supported by the connected EIS instance. If the CCI implementation does not support result set functionality, then the method `getResultSetInfo` is required to throw a `NotSupportedException`.

The method `close` initiates a close of the connection. The OID in Figure 11.0 on page 44 describes the resulting behavior of such an application level connection close.

The method `getLocalTransaction` returns a `LocalTransaction` instance that enables a component to demarcate resource manager local transactions. If a resource adapter does not allow a component to demarcate local transactions using `LocalTransaction` interface, then the method `getLocalTransaction` must throw a `NotSupportedException`.

**Auto Commit**

When a `Connection` is in an auto-commit mode, an `Interaction` (associated with the `Connection`) automatically commits after it has been executed. The auto-commit mode must be off if multiple interactions have to be grouped in a single transaction and committed or rolled back as a unit.

CCI does not provide explicit `set/getAutoCommit` methods on the `Connection` interface. This simplifies application programming model for the transaction management.

A resource adapter is required to manage the auto-commit mode as follows:

- A transactional resource adapter (either at `XATransaction` or `LocalTransaction` level) is required to set the auto-commit mode (for a `Connection` instance participating in the transaction) to off within a transaction. This requirement holds for both container-managed and bean-managed transaction demarcation.
- A transactional resource adapter is required to set the auto-commit mode to on (for `Connection` instances) outside a transaction.

These requirements are independent of whether a transaction is managed as a local or XA transaction. A transactional resource adapter should implement this requirement in an implementation-specific manner.

A non-transactional resource adapter (at `NoTransaction` level) is not required to support the auto-commit mode for `Connection`.

## 9.6 Interaction Interfaces

The following section specifies interfaces that enable a component to drive an interaction (as specified in a specification) with an EIS instance and to demarcate resource manager local transactions.

### 9.6.1 Interaction

The javax.resource.cci.Interaction enables a component to execute EIS functions. An Interaction instance supports the following interactions with an EIS instance:

- An execute method that takes an input Record, output Record and an InteractionSpec. This method executes the EIS function represented by the InteractionSpec and updates the output Record.
- An execute method that takes an input Record and an InteractionSpec. This method implementation executes the EIS function represented by the InteractionSpec and produces the output Record as a return value.

If an Interaction implementation does not support a variant of execute method, the method is required to throw a javax.resource.NotSupportedException.

Refer to section 9.9.2 for details on how input and output records are created and used in the above variants of the execute method.

An Interaction instance is created from a Connection and is required to maintain its association with the Connection instance. The close method releases all resources maintained by the resource adapter for the Interaction. The close of an Interaction instance should not close the associated Connection instance.

The following code extract shows the Interaction interface:

```
public interface javax.resource.cci.Interaction {

     public Connection getConnection();

     public void close() throws ResourceException;

     public boolean execute(InteractionSpec ispec,
                    Record input,
                    Record output) throws ResourceException;

     public Record execute(InteractionSpec ispec,
                    Record input) throws ResourceException;
     // ...
}
```

### 9.6.2 InteractionSpec

A javax.resource.cci.InteractionSpec holds properties for driving an Interaction with an EIS instance. An InteractionSpec uses an Interaction to execute the specified function on an underlying EIS.

The CCI specification defines a set of standard properties for an InteractionSpec. The properties are defined either on a derived interface or an implementation class of an empty InteractionSpec interface. The following code extract shows the InteractionSpec interface.

```
public interface javax.resource.cci.InteractionSpec
             extends java.io.Serializable {

     // Standard Interaction Verbs
     public static final int SYNC_SEND = 0;
     public static final int SYNC_SEND_RECEIVE = 1;
     public static final int SYNC_RECEIVE = 2;

}
```

An InteractionSpec implementation is not required to support a standard property if that property does not apply to its underlying EIS. The InteractionSpec implementation class is required to provide getter and setter methods for each of its supported properties. The getter and setter methods convention should be based on the Java Beans design pattern.

### Standard Properties

The standard properties are as follows:

- FunctionName: A string representing the name of an EIS function. Examples are: name of a transaction program in a CICS system or name of a business object/function module in an ERP system. The format of the name is specific to an EIS and is outside the scope of the CCI specification.
- InteractionVerb: An integer representing the mode of interaction with an EIS instance as specified by the InteractionSpec. The values of interaction verb may be one of the following:
  - SYNC_SEND: The execution of an Interaction does only a send to the target EIS instance. The input record is sent to the EIS instance without any synchronous response in terms of an output Record or ResultSet.
  - SYNC_SEND_RECEIVE: The execution of an Interaction sends a request to the EIS instance and receives response synchronously. The input record is sent to the EIS instance with the output received either as Record or a ResultSet.
  - SYNC_RECEIVE: The execution of an Interaction results in a synchronous receive of an output Record. An example is: a session bean gets a method invocation and it uses this SYNC_RECEIVE form of interaction to retrieve messages that have been delivered to a message queue.

  The default for the InteractionVerb property is SYNC_SEND_RECEIVE.

  If the InteractionVerb property is not defined for an InteractionSpec, then the default mode for an interaction is SYNC_SEND_RECEIVE.

  Other forms of interaction verbs are outside the scope of the CCI specification.

  The CCI does not support asynchronous delivery of messages to the component instances. The EJB 2.0 specification addresses this facility as part of its JMS integration.
- ExecutionTimeout: An integer representing the number of milliseconds an Interaction waits for an EIS to execute the specified function.

### ResultSet Properties

The following standard properties give hints to an Interaction instance about the ResultSet requirements:

- FetchSize: An integer representing the number of rows that should be fetched from an EIS when more rows are needed for a result set. If the value is zero, then the hint is ignored. The default value is zero.
- FetchDirection: An integer representing the direction in which the rows in a result set are processed. The valid integer values are defined in the java.sql.ResultSet interface. The default value is ResultSet.FETCH_FORWARD.
- MaxFieldSize: An integer representing the maximum number of bytes allowed for any value in a column of a result set or a value in a Record.
- ResultSetType: An integer representing the type of the result set produced by an execution of the InteractionSpec. The java.sql.ResultSet interface defines the result set types.
- ResultSetConcurrency: An integer representing the concurrency type of the result set produced by the execution of the InteractionSpec. The java.sql.ResultSet interface defines the concurrency types for a result set.

Note that if a CCI implementation cannot support specified requirements for a result set, then it should choose an appropriate alternative and raise a `SQLWarning` (from the `ResultSet` methods) to indicate this condition. Refer the CCI `ResultSet` interface for more details.

A component can determine the actual scrolling ability and concurrency type of a result set by invoking the methods `getType` and `getConcurrencyType` on the `ResultSet` interface.

#### Additional Properties

An `InteractionSpec` implementation may define additional properties besides the standard properties. Note that the format and type of the additional properties is specific to an EIS and is outside the scope of the CCI specification.

#### Implementation

It is required that the `InteractionSpec` interface be implemented as a JavaBean to support tools. The properties on the `InteractionSpec` implementation class must be defined through the getter and setter methods pattern.

The CCI implementation may (though is not required to) provide a `BeanInfo` class for the `InteractionSpec` implementation. This class provides explicit information about the properties supported by the `InteractionSpec`.

An implementation class for `InteractionSpec` interface is required to implement the `java.io.Serializable` interface.

The specified properties must be implemented as either bound or constrained properties. Refer to the Java Beans specification for details on bound and constrained properties.

#### Administered Object

An `InteractionSpec` instance may be (though it is not required) registered as an administered object in the JNDI namespace. This enables a component provider to access `InteractionSpec` instances using "logical" names called resource environment references. Resource environment references are special entries in the component's environment. The deployer binds a resource environment reference to an `InteractionSpec` administered object in the operational environment.

The EJB 2.0 specification [1] specifies resource environment references in more detail.

#### Illustrative Scenario

The development tool introspects the `InteractionSpec` implementation class and shows a property sheet with all the configurable properties. The developer then configures the properties for an `InteractionSpec` instance.

At run-time, the configured `InteractionSpec` instance is used to specify properties for the execution of an `Interaction`. The run-time environment may lookup an `InteractionSpec` instance using a logical name from the JNDI namespace.

### 9.6.3 LocalTransaction

The `javax.resource.cci.LocalTransaction` defines a transaction demarcation interface for resource manager local transactions. An application component uses `LocalTransaction` interface to demarcate local transactions. Refer chapter 6 for more details on local transactions.

Note that this interface is used for local transaction demarcation at the application level; while the interface `javax.resource.spi.LocalTransaction` is defined as part of the system contracts and is used by a container for local transaction management.

The following code extract shows the `LocalTransaction` interface:

```
public interface javax.resource.cci.LocalTransaction {
     public void begin() throws ResourceException;
     public void commit() throws ResourceException;
```

```
        public void rollback() throws ResourceException;
}
```

**Requirements**

A CCI implementation may (though is not required to) implement the `LocalTransaction` interface.

If the `LocalTransaction` interface is supported by a CCI implementation, then the method `Connection.getLocalTransaction` must return an `LocalTransaction` instance. A component may then use the returned `LocalTransaction` to demarcate a resource manager local transaction on the underlying EIS instance.

A resource adapter is allowed to implement `javax.resource.spi.LocalTransaction` interface without implementing the application-level `javax.resource.cci.LocalTransaction`. In this case, a container uses the system contract level `LocalTransaction` interface for managing local transactions. Refer 6.7 for more details on local transaction management.

## 9.7 Basic Metadata Interfaces

The following section specifies the interfaces that provide basic meta information about a resource adapter implementation and an EIS connection.

### 9.7.1 ConnectionMetaData

The interface `javax.resource.cci.ConnectionMetaData` provides information about an EIS instance connected through a `Connection` instance. A component calls the method `Connection.getMetaData` to get a `ConnectionMetaData` instance.

The following code extract shows the `ConnectionMetaData` interface:

```
public interface javax.resource.cci.ConnectionMetaData {
      public String getEISProductName() throws ResourceException;
      public String getEISProductVersion() throws ResourceException;
      public String getUserName() throws ResourceException;

}
```

The method `getEISProductName` and `getEISProductVersion` return information about the EIS instance.

The method `getUserName` returns the user name for an active connection as known to the underlying EIS instance. The name corresponds the resource principal under whose security context a connection to the EIS instance has been established.

**Implementation**

A CCI implementation is required to provide an implementation class for the `ConnectionMetaData` interface.

A resource adapter provider or third party vendor may extend the `ConnectionMetaData` interface to provide additional information. Note that the format and type of the additional information is specific to an EIS and is outside the scope of the CCI specification.

### 9.7.2 ResourceAdapterMetaData

The interface `javax.resource.cci.ResourceAdapterMetaData` provides information about the capabilities of a resource adapter implementation. Note that this interface does not provide information about an EIS instance that is connected through a resource adapter.

A component uses the ConnectionFactory.getMetaData method to get metadata information about a resource adapter. The getMetaData method does not require that an active connection to an EIS instance should have been established.

The following code extract shows the ResourceAdapterMetaData interface:

```
public interface javax.resource.cci.ResourceAdapterMetaData {
     public String getAdapterVersion();
     public String getAdapterVendorName();
     public String getAdapterName();
     public String getAdapterShortDescription();

     public String getSpecVersion();

     public String[] getInteractionSpecsSupported();
     public boolean supportsExecuteWithInputAndOutputRecord();
     public boolean supportsExecuteWithInputRecordOnly();

     public boolean supportsLocalTransactionDemarcation();
}
```

The method getSpecVersion returns a string representation of the version of the connector architecture specification that is supported by the resource adapter.

The method getInteractionSpecsSupported returns an array of fully-qualified names of InteractionSpec types supported by the CCI implementation for this resource adapter. Note that the fully-qualified class name is for the implementation class of an InteractionSpec. This method may be used by tools vendor to find information on the supported InteractionSpec types. The method should return an array of length 0 if the CCI implementation does not define specific InteractionSpec types.

The methods supportsExecuteWithInputAndOutputRecord and supportsExecuteWithInputRecordOnly are used by tools vendor to find information about the Interaction implementation. It is important to note that Interaction implementation must support atleast one variant of execute methods.

The method supportsExecuteWithInputAndOutputRecord returns true if the implementation class for the Interaction interface implements public boolean execute(InteractionSpec ispec, Record input, Record output) method; otherwise the method returns false.

The method supportsExecuteWithInputRecordOnly returns true if the implementation class for the Interaction interface implements public Record execute(InteractionSpec ispec, Record input) method; otherwise the method returns false.

The method supportsLocalTransactionDemarcation returns true if the resource adapter implements the LocalTransaction interface and supports local transaction demarcation on the underlying EIS instance through the LocalTransaction interface.

The ResourceAdapterMetaData may be extended to provide more information specific to a resource adapter implementation.

## 9.8 Exception Interfaces

The following section specifies ResourceException class defined by the CCI,

### 9.8.1 ResourceException

The javax.resource.ResourceException class is used as the root of the exception hierarchy for CCI. A ResourceException provides the following information:

- A resource adapter-specific string describing the error. This string is a standard Java exception message and is available through the `getMessage` method.
- A resource adapter-specific error code
- A reference to another exception. A `ResourceException` is often the result of a lower level problem. If appropriate, this lower level exception (a `java.lang.Exception` or its derived exception type) can be linked to a `ResourceException` instance.

A CCI implementation can extend the `ResourceException` interface to throw more specific exceptions. It may also chain instances of `java.lang.Exception` or its subtypes to a `ResourceException`.

### 9.8.2 ResourceWarning

The `javax.resource.cci.ResourceWarning` class provides information on the warnings related to interactions with EIS. A `ResourceWarning` is silently chained to an `Interaction` instance that has caused the warning to be reported.

The methods `Interaction.getWarnings` enable a component to access the first `ResourceWarning` in a chain of warnings. Other `ResourceWarning` instances are chained to the first returned `ResourceWarning` instance.

## 9.9 Record

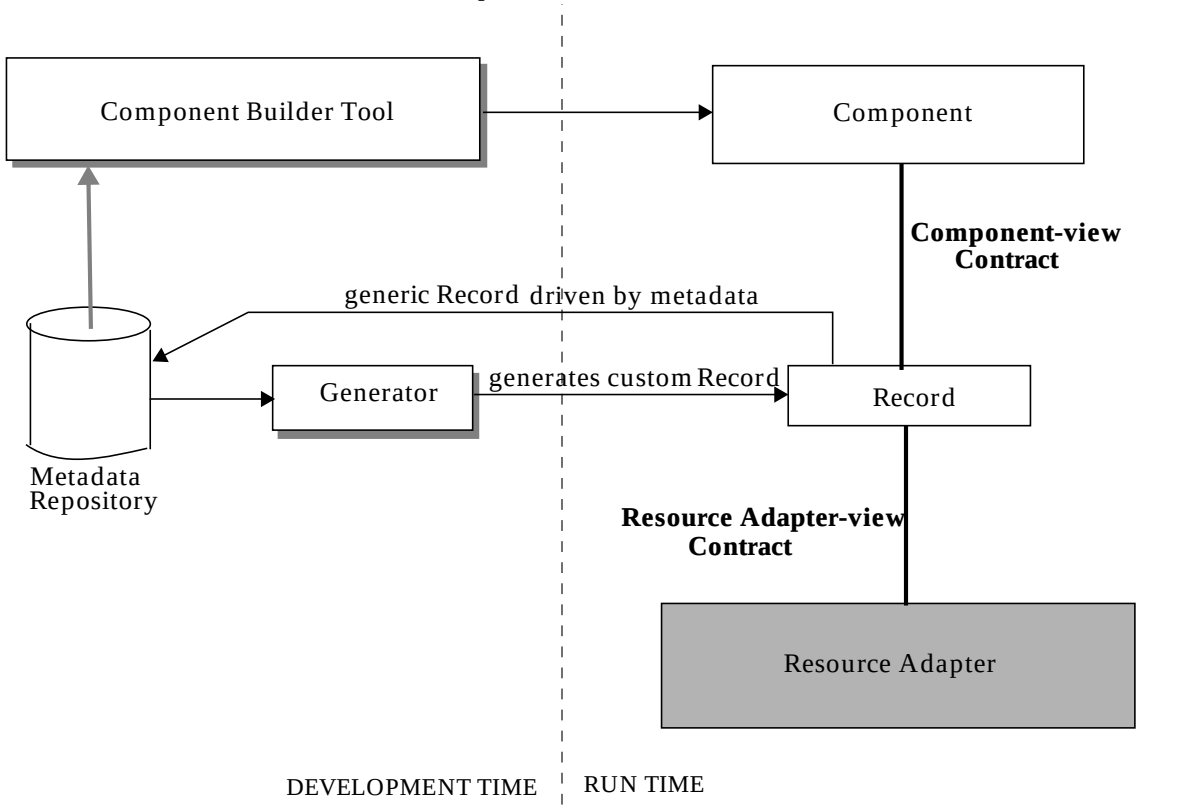
A `Record` is the Java representation of a data structure used as input or output to an EIS function.

A `Record` has both development-time and run-time aspects. See Figure 33.0. An implementation of a `Record` is either:

- A custom `Record` implementation that gets generated at the development time by a tool. The generation of a custom implementation is based on the meta information accessed by the tool from a metadata repository. The type mapping and data representation is generated as part of the custom `Record` implementation. So the custom `Record` implementation typically does not need to access the metadata repository at run-time.
- A generic `Record` implementation that uses a metadata repository at run-time for meta information. For example, a generic type of `Record` may access the type mapping information from the repository at run-time.

**Note**: The specification of a standard repository APIs and metadata format is outside the scope of the current version 1.0 of the connector architecture.

**FIGURE 33.0** Record at Development-time and Run-time




The meta information used in a `Record` representation and type mapping may be available in a metadata repository as:

- Meta information expressed in an EIS specific format. For example, an ERP system has its own descriptive format for its meta information.
- Formatted based on the programming language that has been used for writing the target EIS function. For example, `COBOL` structures used by CICS transaction programs.
- Standard representation of data structures as required for EIS functions. The standard representation is typically aggregated in a metadata repository based on the meta information extracted from multiple EISs.

A resource adapter may provide an implementation of all CCI interfaces except the data representation-related interfaces—namely, `Record` and `RecordFactory`. In this case, a third party vendor provides both development-time and run-time support for `Record` and `RecordFactory` interfaces. This requires that a `Record` implementation must support both component-view and resource adapter-view contracts, as specified in the following subsections.

### 9.9.1 Component-view Contract

The component-view contract provides a standard contract in terms of using a `Record` for components and component building tools. A `Record` implementation is required to support the component-view contract.

The application programming model for a `Record` is as follows:

- A component creates an instance of a generated implementation class for a custom record. The implementation class represents an EIS specific data structure.

- A component uses the `RecordFactory` interface (refer the component-view contract) to create an instance of the generic `Record` implementation class. The implementation class of a generic `Record` is independent of any EIS-specific data structure.

---

**Note**: A CCI related issue is the level of support in the CCI data representation interfaces (namely, `Record`, `MappedRecord` and `IndexedRecord`) for the type mapping facility. The issue has to be addressed based on the following parameters:

- There is no standardized mapping across various type systems. For example, the existing type systems range from Java, `CORBA`, `COM`, `COBOL` and many more. It is difficult to standardize the type specification and mappings across such a diverse set of type systems within the connector architecture 1.0 scope.
- Building a limited type mapping facility into the CCI data representation interfaces will constrain the use of CCI data representation interfaces across different types of EISs. For example, it may be difficult to support EISs that have complex structured types with a limited type mapping support.
- Building an extensive type mapping facility into the present version 1.0 CCI data representation interfaces will limit the future extensibility of these interfaces. This applies specifically to the support for standards that are emerging for XML-based data representation. An important goal for CCI data representation interfaces is to support XML-based facilities. This goal is difficult to achieve in 1.0 scope of the connector architecture.

The specification proposes that the type mapping support for the CCI be kept open for future versions. The connectors.next (or a separate JSR) may also focus on standardizing type mappings.

---

### Type Mapping

Type mapping for EIS-specific types to Java types is not directly exposed to an application component. For example in case of a custom `Record` implementation, the getter and setter methods (defined on a `Record` and exposed to an application component) return the correct Java types for the values extracted from the `Record`. The custom `Record` implementation internally handles all the type mapping.

In case of a generic `Record` implementation, the type mapping is done in the generic `Record` by means of the type mapping information obtained from the metadata repository. Since the component uses generic methods on the `Record` interface, the component code does the required type casting.

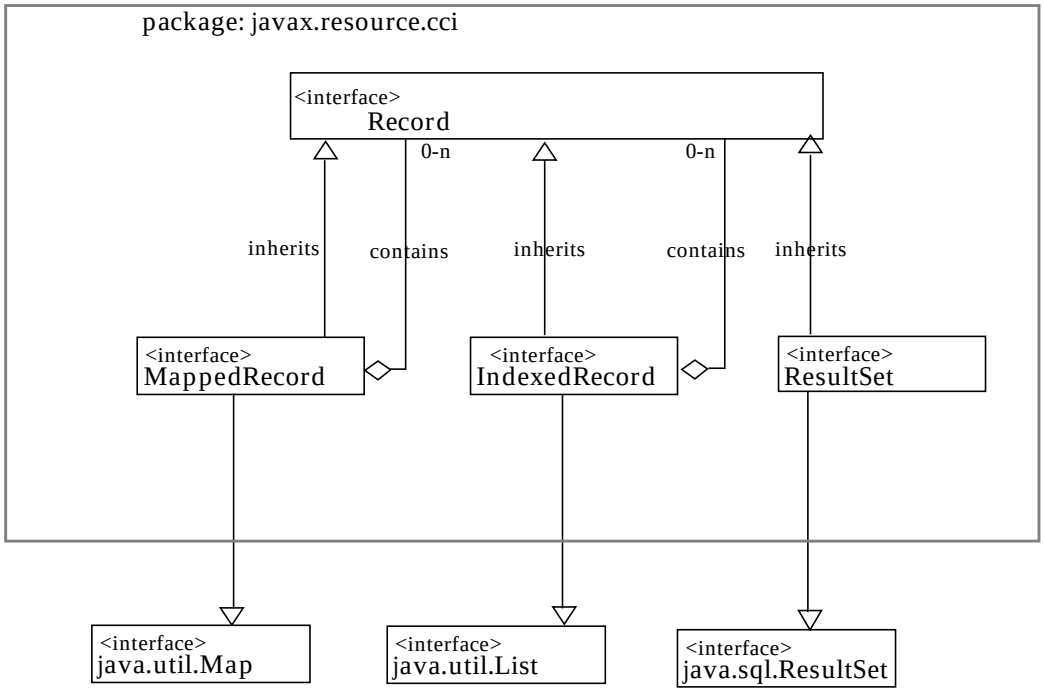
The compatibility of Java types and EIS types should be based on a type mapping that is defined specific to a *class* of EISs. For example, an ERP system from vendor X specifies a type mapping specific to its own EIS. Another example is type mapping between Java and `COBOL` types. Note that the JDBC specification specifies a standard mapping of SQL data types to Java types specific to relational databases.

In cases of both custom and generic `Records`, the type mapping information is provided by a metadata repository either at development-time or run-time.

### Record Interface

The `javax.resource.cci.Record` interface is the base interface for the representation of a record. A `Record` instance is used as an input or output to the `execute` methods defined on an `Interaction`. See Figure 34.0.

**FIGURE 34.0** Component-view Contract




The `Record` interface may be extended to form one of the following representations:

- `javax.resource.cci.MappedRecord`: A key-value pair based collection represents a record. This interface is based on the `java.util.Map`.
- `javax.resource.cci.IndexedRecord`: An ordered and indexed collection represents a record. This interface is based on the `java.util.List`.
- `javax.resource.cci.ResultSet`: This interface extends both `java.sql.ResultSet` and `javax.resource.cci.Record`. A result set represents tabular data. The section 9.10 specifies the requirements for the `ResultSet` interface in detail.
- JavaBean based representation of an EIS data structure: An example is a custom record generated to represent a purchase order in an ERP system or an invoice in a mainframe TP system.

Refer to Section 9.11 for code samples that illustrate the use of record.

`MappedRecord` or `IndexedRecord` may contain another `Record`. This means that `MappedRecord` and `IndexedRecord` can be used to create a hierarchical structure of any arbitrary depth.

`MappedRecord` and `IndexedRecord` can be used to represent either generic or custom record.

A basic Java type is used as the leaf element of a hierarchical structure represented by a `MappedRecord` or `IndexedRecord`.

A generated custom `Record` may also contain other records to form a hierarchical structure.

The following code extract shows the `Record` interface:

```
public interface javax.resource.cci.Record
            extends java.lang.Cloneable, java.io.Serializable {

     public String getRecordName();
```

```
        public void setRecordName(String name);

        public void setRecordShortDescription(String description);
        public String getRecordShortDescription();

        public boolean equals(Object other);
        public int hashCode();

        public Object clone() throws CloneNotSupportedException;
    }
```

The Record interface defines the following set of standard properties:

- Name of a Record: Note that the CCI does not define a standard format for naming a Record. The name format is specific to an EIS type.
- Description of a Record: This property is used primarily by tools to show a description of a Record instance.

### MappedRecord and IndexedRecord Interfaces

The javax.resource.cci.MappedRecord interface is used for representing a key-value map based collection of record elements. The MappedRecord interface extends both Record and java.util.Map interface.

```
public interface javax.resource.cci.MappedRecord
                  extends Record, java.util.Map, java.io.Serializable {
}
```

The javax.resource.cci.IndexedRecord interface represents an ordered collection of record elements based on the java.util.List interface. This interface allows a component to access record elements by their integer index (position in the list) and search for elements in the list.

```
public interface javax.resource.cci.IndexedRecord
                  extends Record, java.util.List, java.io.Serializable {
}
```

### RecordFactory

The javax.resource.cci.RecordFactory interface is used for creating MappedRecord and IndexedRecord instances. Note that the RecordFactory is only used for creation of generic record instances. A CCI implementation provides an implementation class for the RecordFactory interface.

The following code extract shows the RecordFactory interface:

```
public interface javax.resource.cci.RecordFactory {
     public MappedRecord createMappedRecord(String recordName)
                  throws ResourceException;

     public IndexedRecord createIndexedRecord(String recordName)
                  throws ResourceException;

}
```

The methods `createMappedRecord` and `createIndexedRecord` take the name of the record that is to be created by the `RecordFactory`. The name of the record acts as a pointer to the meta information (stored in the metadata repository) for a specific record type. The format of the name is outside the scope of the CCI specification and specific to a CCI implementation and / or metadata repository.

A `RecordFactory` implementation should be capable of using the name of the desired `Record` and accessing meta information for the creation of the `Record`.

### 9.9.2 Interaction and Record

Records should be used as follows for the two variants of the `execute` method on the `Interaction` interface:

Method: `boolean execute(InteractionSpec, Record input, Record output)`:

- A custom record instance is used as an input or output to the `execute` method. A custom record implementation class is generated by an application development tool or EAI framework based on the meta information.
- The `RecordFactory` interface is used to create a generic `MappedRecord` or `IndexedRecord` instance. The generic record is used as input or output to the `execute` method.

Method: `Record execute(InteractionSpec, Record input)`:

- The input record can be either a custom or generic record.
- The returned record is a generic record instance created by the implementation of the `execute` method. The generic record instance may represent `ResultSet` or an hierarchical structure (as represented through the `MappedRecord` and `IndexedRecord` interfaces).

When `Interaction.execute` method is called, a generic record instance may use the connection (associated with the `Interaction` instance) to access the metadata from the underlying EIS. If there is a separate metadata repository, then generic record gets the metadata from the repository. The generic record implementation may use the above illustrative mechanism to achieve the necessary type mapping.

The generic record implementation encapsulates the above behavior and interacts with `Interaction` implementation in the `execute` method to get the active connection; if so needed. The contract between generic record and `Interaction` implementation classes is specific to a CCI implementation.

### 9.9.3 Resource Adapter-view Contract

A resource adapter views the data represented by a `Record` either as:

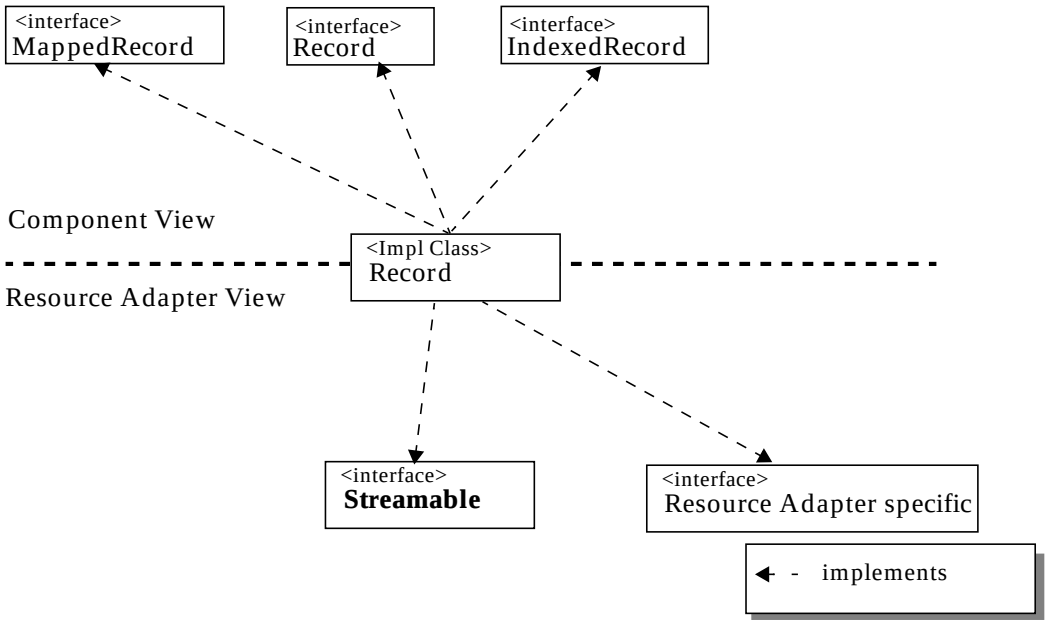
- A stream of bytes through the `Streamable` interface, or,
- A format specific to a resource adapter. For example, a resource adapter may extract (or set) the data for a `Record` using an interface defined specific to the resource adapter.

A resource adapter-specific interface for viewing `Record` representation is outside the scope of the CCI specification. A resource adapter is required to describe the resource adapter-specific interface to the users (tools vendors) of the resource adapter-view contract.

#### Streamable Interface

The `javax.resource.cci.Streamable` interface enables a resource adapter to extract data from an input `Record` or set data into an output `Record` as a stream of bytes. See Figure 35.0.

**FIGURE 35.0** Streamable Interface




The `Streamable` interface provides a resource adapter's view of the data set in a `Record` instance by a component. A component uses `Record` or any derived interfaces to manage records.

A component does not directly use the `Streamable` interface. The interface is used by a resource adapter implementation.

The following code extract shows the `Streamable` interface:

```
public interface javax.resource.cci.Streamable {
     public void read(InputStream istream) throws IOException;
     public void write(OutputStream ostream) throws IOException;
}
```

The method `read` extracts data from an `InputStream` and initializes fields of a `Streamable` object. The method `write` writes fields of a `Streamable` object to an `OutputStream`. The implementations of both `read and write` methods for a `Streamable` object must call the `read` and `write` methods respectively on the super class if there is one.

An implementation class of `Record` may choose to implement the `Streamable` interface or support a resource adapter specific interface to manage record data.

## 9.10 ResultSet

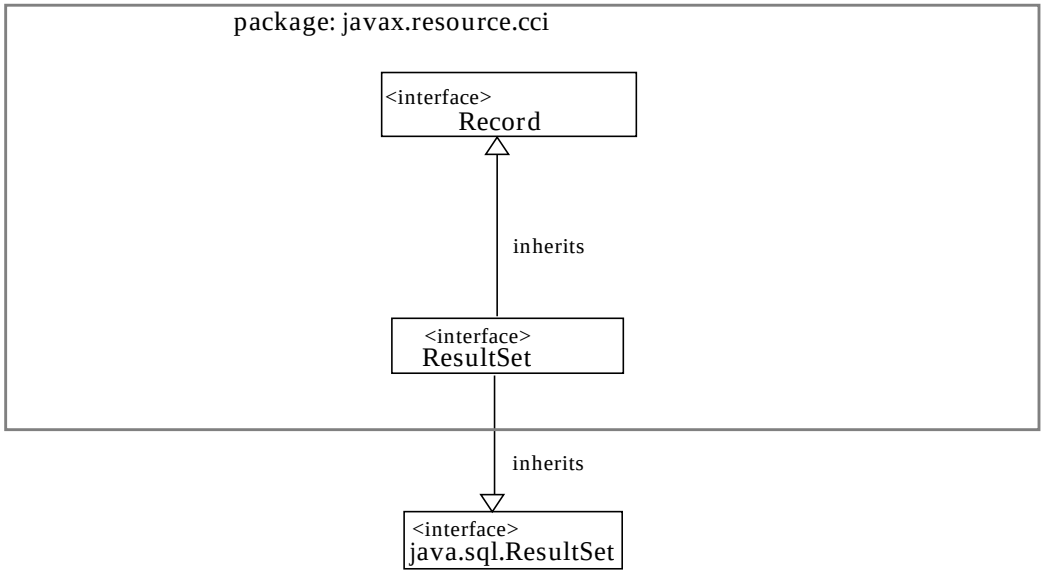
A result set represents tabular data that is retrieved from an EIS instance by the execution of an interaction. The method `execute` on the `Interaction` interface can return a `ResultSet` instance.

The CCI `ResultSet` interface is based on the JDBC `ResultSet` interface. The `javax.resource.cci.ResultSet` extends the `java.sql.ResultSet` and `javax.resource.cci.Record` interfaces.

The following code extract shows the `ResultSet` interface:

```
public interface javax.resource.cci.ResultSet
                 extends Record, java.sql.ResultSet {
}
```

**FIGURE 36.0** ResultSet interface




The following section specifies requirements for a CCI `ResultSet` implementation.

Refer to the JDBC [3] specification and Java docs for more details on the `java.sql.ResultSet` interface. The following section specifies only a brief outline of the `ResultSet` interface. It focuses on the differences between the implementation requirements set by the CCI and JDBC. Note that the JDBC semantics for a `ResultSet` hold for the cases that are not explicitly mentioned in the following section.

The decision to use the JDBC `ResultSet` for the CCI has been taken because of the following reasons:

- JDBC `ResultSet` is a standard, established, and well-documented interface for accessing and updating tabular data.
- JDBC `ResultSet` interface is defined in the core `java.sql package`. An introduction of an independent CCI-specific `ResultSet` interface (that is different from the JDBC `ResultSet` interface) may create confusion in terms of differences in the programming model and functionality.
- The use of the JDBC `ResultSet` interface enables a tool or EAI vendor to leverage existing facilities that have been built over the JDBC `ResultSet`.

**Important**: A CCI implementation is not required to support `javax.resource.cci.ResultSet` interface. If a CCI implementation does not support result set functionality, then it should not support interfaces and methods that are associated with the result set functionality. An example is the `java.sql.ResultSetMetaData` interface.

### 9.10.1 ResultSet Interface

The `ResultSet` interface provides a set of `getXXX` methods for retrieving column values from the current row. A column value can be retrieved using either the index number of the column

or the name of the column. The columns are numbered starting from one. For maximum portability, result set columns within each row should be read in a left-to-right order, and each column should be read only once.

The `ResultSet` interface also defines a set of `updateXXX` methods for updating the column values of the current row.

### Type Mapping

A `ResultSet` implementation should attempt to convert the underlying EIS specific data type to the Java type (as specified in the `XXX` part) of the `getXXX` method and return a suitable Java value.

A `ResultSet` implementation is required to establish a type mapping between the EIS specific data types and Java types. The type mapping is specific to an EIS.

The CCI specification does not specify standard type mappings specific to each type of EIS.

### ResultSet Types

The CCI `ResultSet` (similar to the JDBC `ResultSet`) supports the following types of result set: `forward-only`, `scroll-insensitive` and `scroll-sensitive`.

A forward-only result set is non-scrollable; its cursor moves only forward, from top to bottom. The view of data in the result set depends on whether the EIS instance materializes results incrementally.

A scroll-insensitive result set is scrollable; its cursor can move forward or backward and can be moved to a particular row or to a row whose position is relative to the current row. This type of result set is not sensitive to any changes (made by another transaction or result sets in the same transaction) that are made while the result set is open. This type of result set provides a static view of the underlying data with respect to changes made by other result sets. The order and values of rows are set at the time of the creation of a scroll-insensitive result set.

A scroll-sensitive result set is scrollable. It is sensitive to changes that are made while the result set is open. This type of a result set provides more dynamic view of the underlying data.

A component can use methods `ownUpdatesAreVisible`, `ownDeletesAreVisible` and `ownInsertsAreVisible` on the `ResultSetInfo` interface to determine whether a result set can see its own changes while the result set is open. For example, a result set own changes are visible if the updated column values can be retrieved by calling the `getXXX` method after the corresponding `updateXXX` method. Refer to the JDBC [3] specification for more details on this aspect.

### Scrolling

The CCI `ResultSet` supports the same scrolling ability as the JDBC `ResultSet`.

If a resource adapter implements the cursor movement methods, then its result sets are scrollable. A scrollable result set created by executing an `Interaction` can move through its contents in both forward (first-to-last) or backward (last-to-first) direction. A scrollable result set also supports relative and absolute positioning.

The CCI `ResultSet` (similar to the JDBC `ResultSet`) maintains a cursor that indicates the row in the result set that is currently being accessed. The cursor maintained on a `forward-only` result set can only move forward through the contents of the result set. The rows are accessed in a first-to-last order. A scrollable result set can also be moved in a backward direction (first-to-last) and to a particular row.

Note that a CCI `ResultSet` implementation should only provide support for scrollable result sets if the underlying EIS supports such a facility.

### Concurrency Types

A component can set the concurrency type of a CCI `ResultSet` to be either read-only or updatable. These types are consistent with the concurrency types defined by the JDBC `ResultSet`.

A result set that uses read-only concurrency does not allow updates of its content, while an updatable result set allows updates to its contents. An updatable result set may hold a write lock on the underlying data item and thus reduce concurrency.

Refer to the JDBC [3] for detailed specification and examples.

### Updatability

A result set of concurrency type `CONCUR_UPDATABLE` supports update, insert and delete of its rows. The CCI support for this type of result set is similar to the JDBC `ResultSet`.

The methods `updateXXX` on the `ResultSet` interface are used to modify the values of an individual column in the current row. These methods do not update the underlying EIS. The method `updateRow` must be called to update data on the underlying EIS. A resource adapter may discard changes made by a component if the component moves the cursor from the current row before calling the method `updateRow`.

Refer to the JDBC [3] specification for detailed specification and examples.

### Persistence of Java Objects

The `ResultSet` interface provides the `getObject` method to enable a component to retrieve column values as Java objects. The type of the Java object returned from the `getObject` method is compatible to the type mapping supported by a resource adapter specific to its underlying EIS. The `updateObject` method enables a component to update a column value using a Java object.

### Support for SQL Types

It is optional for a CCI `ResultSet` to support the `SQL` type `JAVA_OBJECT` (as defined in the `java.sql.Types`).The JDBC [3] specification specifies the JDBC support for persistence of Java objects.

The support for the following `SQL` types (as defined in the `java.sql.Types`) is optional for a CCI `ResultSet` implementation:

- Binary large object (`BLOB`)
- Character large object (`CLOB`)
- `SQL ARRAY` type
- `SQL REF` type
- `SQL DISTINCT` type
- `SQL STRUCT` type

If an implementation of the CCI `ResultSet` interface does not support these types, it is required to throw a `java.sql.SQLException` (indicating that the method is not supported) or `java.lang.UnsupportedOperationException` from the following methods:

- `getBlob`, `getClob`, `getArray`, `getRef`

### Support for Customized SQL Type Mapping

The CCI is not required to support customized mapping of SQL structured and distinct types to Java classes. The JDBC API defines support for such customization mechanisms.

The CCI `ResultSet` should throw a `java.sql.SQLException` (indicating that the method is not supported) or `java.lang.UnsupportedOperationException` from the `getObject` method that takes a `java.util.Map` parameter.

### 9.10.2 ResultSetMetaData

The interface `java.sql.ResultSetMetaData` provides information about the columns in a `ResultSet` instance. A component uses `ResultSet.getMetaData` method to get information about a `ResultSet`.

Refer JDBC Javadocs for a detailed specification of the `ResultSetMetaData` interface.

### 9.10.3 ResultSetInfo

The interface `javax.resource.cci.ResultSetInfo` provides information on the support provided for `ResultSet` functionality by a connected EIS instance. A component calls the method `Connection.getResultInfo` to get the `ResultSetInfo` instance.

A CCI implementation is not required to support `javax.resource.cci.ResultSetInfo` interface. The implementation of this interface is provided only if the CCI supports the `ResultSet` facility.

The following code extract shows the `ResultSetInfo` interface:

```
public interface javax.resource.cci.ResultSetInfo {
     public boolean updatesAreDetected(int type)
                                    throws ResourceException;
     public boolean insertsAreDetected(int type)
                                    throws ResourceException;
     public boolean deletesAreDetected(int type)
                                    throws ResourceException;

     public boolean supportsResultSetType(int type)
                                    throws ResourceException;
     public boolean supportsResultTypeConcurrency(int type,
                                    int concurrency)
                                    throws ResourceException;

     public boolean ownUpdatesAreVisible(int type)
                                    throws ResourceException;
     public boolean ownInsertsAreVisible(int type)
                                    throws ResourceException;
     public boolean ownDeletesAreVisible(int type)
                                    throws ResourceException;


     public boolean othersUpdatesAreVisible(int type)
                                    throws ResourceException;
     public boolean othersInsertsAreVisible(int type)
                                    throws ResourceException;
     public boolean othersDeletesAreVisible(int type)
                                    throws ResourceException;
}
```

The type parameter to the above methods represents the type of the `ResultSet` —defined as `TYPE_XXX` in the `ResultSet` interface.

Note that these methods should throw a `ResourceException` in the following cases:

- A resource adapter and the connected EIS instance cannot provide any meaningful values for these properties.
- The CCI implementation does not support the `ResultSet` functionality. In this case, `NotSupportedException` should be thrown from invocations on the above methods.

A component uses the methods `rowUpdated`, `rowInserted`, and `rowDeleted` on the `ResultSet` interface to determine whether a row has been affected by a visible update, insert, or delete is the result set is open. The methods `updatesAreDetected`, `insertsAreDetected` and `deletesAreDetected` enable a component to find whether or not changes to a `ResultSet` are detected.

A component uses the methods `ownUpdatesAreVisible`, `ownDeletesAreVisible` and `ownInsertsAreVisible` interface to determine whether a `ResultSet` can see its own changes when the result set is open.

A component uses the method `supportsResultSetType` to check the types of `ResultSet` types supported by a resource adapter and its underlying EIS instance.

The method `supportsResultSetTypeConcurency` provides information on the `ResultSet` concurrency types supported by a resource adapter and its underlying EIS instance.

## 9.11 Code Samples

The following code extracts illustrate the application programming model based on the CCI.

An application development tool or EAI framework normally hides all the CCI-based programming details from an application developer. For example, an application development tool generates a set of Java classes that abstract the CCI-based application programming model and offers a simple programming model to an application developer.

### 9.11.1 Connection

- Get a `Connection` to an EIS instance after lookup of a `ConnectionFactory` instance from the JNDI namespace. In this case, the component allows container to manage the EIS sign-on.

```
javax.naming.Context nc = new InitialContext();
javax.resource.cci.ConnectionFactory cf = (ConnectionFactory)nc.lookup(
                    "java:comp/env/eis/ConnectionFactory");
javax.resource.cci.Connection cx = cf.getConnection();
```

- Create an `Interaction` instance:

```
javax.resource.cci.Interaction ix = cx.createInteraction();
```

### 9.11.2 InteractionSpec

- Create a new instance of the respective `InteractionSpec` class or lookup a pre-configured `InteractionSpec` in the run-time environment using the JNDI.

```
com.wombat.cci.InteractionSpecImpl ixSpec = // ...

ixSpec.setFunctionName("<EIS_SPECIFIC_FUNCTION_NAME>");
ixSpec.setInteractionVerb(InteractionSpec.SYNC_SEND_RECEIVE);
...
```

### 9.11.3 Mapped Record

- Get a `RecordFactory` instance:

```
javax.resource.cci.RecordFactory rf = // ... get a RecordFactory
```

- Create a generic `MappedRecord` using the `RecordFactory` instance. This record instance acts as an input to the execution of an interaction. The name of the `Record` acts as a pointer to the meta information (stored in the metadata repository) for a specific record type.

```
javax.resource.cci.MappedRecord input =
     rf.createMappedRecord(“<NAME_OF_RECORD”);
```

- Populate the generic `MappedRecord` instance with input values. The component code adds values based on the meta information it has accessed from the metadata repository.

```
input.put("<key: element1>", new String("<VALUE>"));
input.put("<key: element2>", ...);
...
```

- Create a generic `IndexedRecord` to hold output values that are set by the execution of the interaction.

```
javax.resource.cci.IndexedRecord output =
     rf.createIndexedRecord("<NAME_OF_RECORD");
```

- Execute the `Interaction`:

```
boolean ret = ix.execute(ixSpec, input, output);
```

- Extract data from the output `IndexedRecord`. Note that the type mapping is done in the generic `IndexedRecord` by means of the type mapping information in the metadata repository. Since the component uses generic methods on the `IndexedRecord`, the component code does the required type casting.

```
java.util.Iterator iterator = output.iterator();
while (iterator.hasNext()) {
     // Get a record element and extract value
}
```

### 9.11.4 ResultSet

- Set the requirements for the `ResultSet` returned by the execution of an `Interaction`. This step is optional; default values are used if requirements are not explicitly set:

```
com.wombat.cci.InteractionSpecImpl ixSpec = // .. get an InteractionSpec;

ixSpec.setFetchSize(20);
ixSpec.setResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE);
```

- Execute an `Interaction` that returns a `ResultSet`:

```
javax.resource.cci.ResultSet rs = (javax.resource.cci.ResultSet)
                                       ix.execute(ixSpec, input);
```

- Iterate over the `ResultSet`. The example here positions the cursor on the first row and then iterates forward through the contents of the `ResultSet`. The `getXXX` methods are used to retrieve column values:

```
rs.beforeFirst();
while (rs.next()) {
     // get the column values for the current row using getXXX method
}
```

- The following example shows a backward iteration through the `ResultSet`:

```
rs.afterLast();
while (rs.previous()) {
     // get the column values for the current row using getXXX method
}
```

### 9.11.5 Custom Record

- Extend the `Record` interface to represent an EIS-specific custom Record. The interface `CustomerRecord` supports a simple getter-setter design pattern for its field values. A development tool generates the implementation class of the `CustomerRecord`.

```
public interface CustomerRecord extends javax.resource.cci.Record,
```

```
                    javax.resource.cci.Streamable {

        public void setName(String name);
        public void setId(String custId);
        public void setAddress(String address);

        public String getName();
        public String getId();
        public String getAddress();
    }
```

- Create an empty `CustomerRecord` instance to hold output from the execution of an `Interaction`.

```
CustomerRecord customer = // ... create an instance
```

- Create a `PurchaseOrderRecord` instance as an input to the `Interaction` and set properties on this instance. The `PurchaseOrderRecord` is another example of a custom `Record`.

```
PurchaseOrderRecord purchaseOrder = // ... create an instance
purchaseOrder.setProductName(“...”);
purchaseOrder.setQuantity(“...”);
// ...
```

- Execute an `Interaction` that populates the output `CustomerRecord` instance.

```
// Execute the Interaction
boolean ret = ix.execute(ixSpec, purchaseOrder, customer);

// Check the CustomerRecord
System.out.println( customer.getName() + ":" +
                    customer.getId() + ":" +
                    customer.getAddress());
```

# 10 Packaging and Deployment

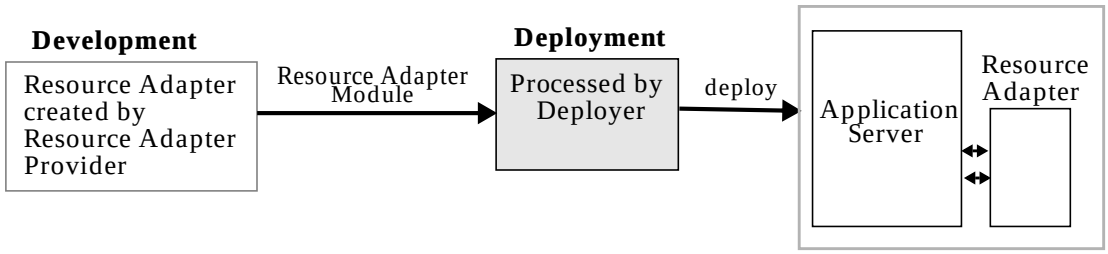
This chapter specifies requirements for packaging and deploying a resource adapter. These requirements support a modular, portable deployment of a resource adapter into a J2EE compliant application server.

## 10.1 Overview

A resource adapter provider develops a set of Java interfaces/classes as part of its implementation of a resource adapter. These Java classes implement connector architecture-specified contracts and EIS-specific functionality provided by the resource adapter. The development of a resource adapter can also require use of native libraries specific to the underlying EIS.

The Java interfaces/classes are packaged together (with required native libraries, help files, documentation, and other resources) with a deployment descriptor to create a `Resource Adapter Module`. A deployment descriptor defines the contract between a resource adapter provider and a deployer for the deployment of a resource adapter.

**FIGURE 37.0** Packaging and Deployment lifecycle of a resource adapter



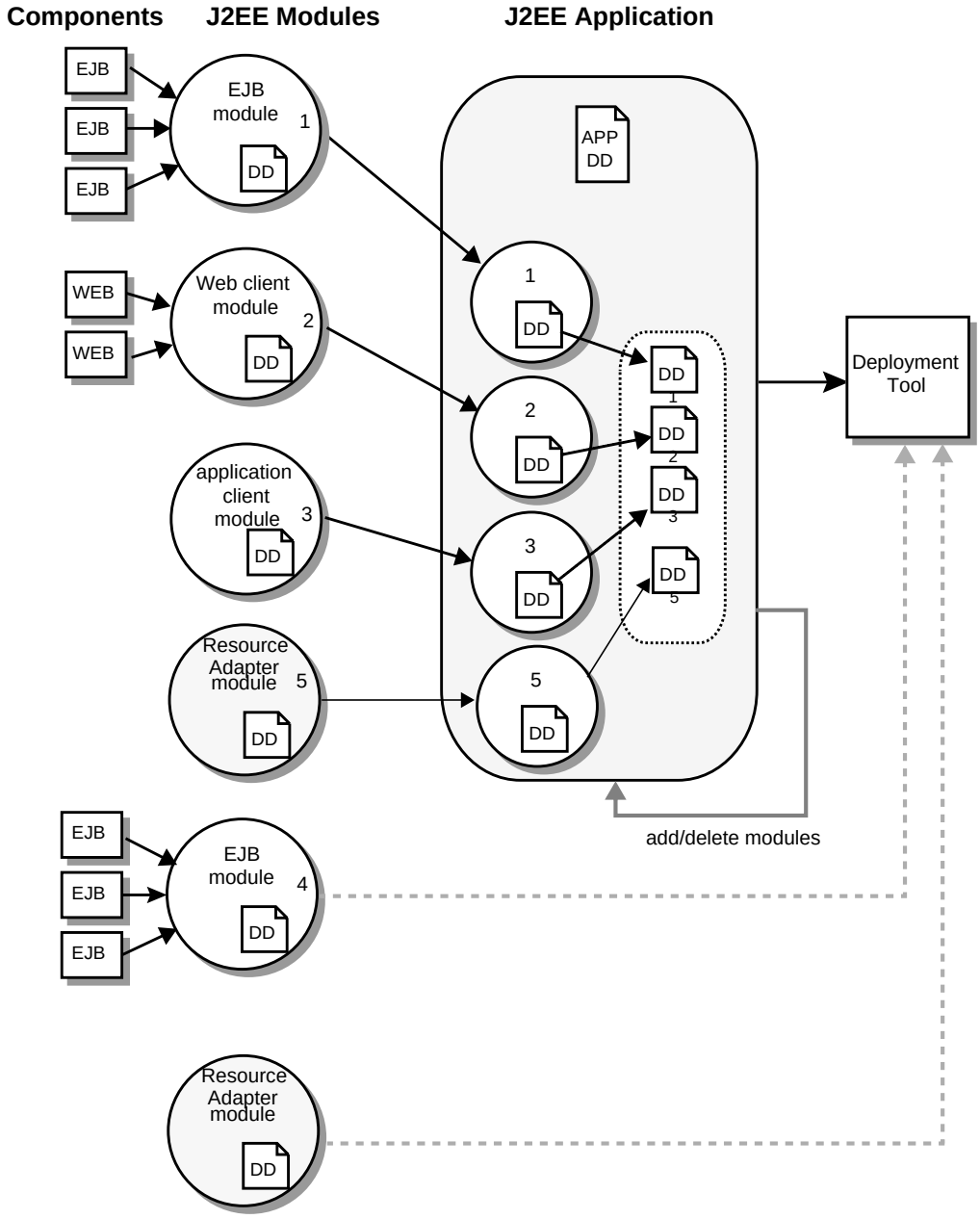

A resource adapter module corresponds to a J2EE module in terms of the J2EE composition hierarchy [refer to the J2EE Platform specification [8] for more details on the deployment of J2EE modules and applications]. A J2EE module represents the basic unit of composition of a J2EE application. Examples of J2EE modules include: EJB module, application client module, and web client module.

A resource adapter module must be deployed either:

- Directly into an application server as a stand-alone unit, or,
- Deployed with a J2EE application that consists of one or more J2EE modules in addition to a resource adapter module. The J2EE specification specifies requirements for the assembly and packaging of J2EE applications.

Figure 38.0 shows the composition model of a resource adapter module with other J2EE modules.

**FIGURE 38.0** Deployment of Resource Adapter module




The stand-alone deployment of a resource adapter module into an application server is typically done to support scenarios in which multiple J2EE applications share a single resource adapter module. However, in certain scenarios, a resource adapter module is required only by components within a single J2EE application. The deployment option of a resource adapter module bundled with a J2EE application supports the latter scenario.

During deployment, the deployer installs a resource adapter module on an application server and then configures it into the target operational environment.

## 10.2 Packaging

The file format for a packaged resource adapter module defines the contract between a resource adapter provider and deployer.

A packaged resource adapter includes the following elements:

- Java classes and interfaces that are required for the implementation of both the connector architecture contracts and the functionality of the resource adapter.
- Utility Java classes for the resource adapter.
- Platform-dependent native libraries required by the resource adapter.
- Help files and documentation.
- Descriptive meta information that ties the above elements together.

### Resource Adapter Archive (RAR)

A resource adapter must be packaged using the Java ARchive (JAR) format in to an RAR (ResourceAdapter ARchive). For example, a resource adapter for EIS A can be packaged as an archive with a filename `eisA.rar`.

The RAR file must contain a deployment descriptor based on the format specified in Section 10.6. The deployment descriptor must be stored with the name `META-INF/ra.xml` in the RAR file.

The Java interfaces, implementation and utility classes (required by the resource adapter) must be packaged as one or more JAR files as part of the resource adapter module. A JAR file is required to use the `.jar` file extension. Java classes packaged using a non `.jar` extension should be handled in a deployment tool-specific way.

The platform-specific libraries required by the resource adapter must be packaged with the resource adapter module.

### Sample Directory Structure

The following illustrates a listing of files in a sample resource adapter module:

```
/META-INF/ra.xml
/howto.html
/images/icon.jpg
/ra.jar
/cci.jar
/win.dll
/solaris.so
```

In the above example, `ra.xml` is the deployment descriptor. The `ra.jar` and `cci.jar` contain Java interfaces and implementation classes for the resource adapter. The `win.dll` and `solaris.so` are examples of native libraries.

Note that a resource adapter module can be structured such that various elements are partitioned using subdirectories.

### Requirements

The deployer must ensure that all the JAR files (packaged within a resource adapter module) are loaded in the operational environment. In addition, the deployer must resolve any dependencies of a resource adapter on platform-specific native libraries. Note that mechanisms for configuring platform-specific libraries in an operational environment are out of scope for the connector architecture.

When a standalone resource adapter RAR is deployed, the resource adapter must be made available to all J2EE applications in the application server.

When a resource adapter RAR packaged within a J2EE application EAR is deployed, the resource adapter must be made available only to the J2EE application with which it is packaged.

A deployment tool is allowed to offer additional deployment choices, which are outside the scope of this specification.

## 10.3 Deployment

A deployment descriptor defines the contract between a resource adapter provider and a deployer. It captures the declarative information that is intended for the deployer to enable deployment of a resource adapter in a target operational environment.

A resource adapter module must be deployed based on the deployment requirements specified by the resource adapter provider in the deployment descriptor. Section 10.6 specifies the XML DTD for the deployment descriptor for a resource adapter module.

### 10.3.1 Resource Adapter Provider

The resource adapter provider is responsible for specifying the deployment descriptor for a resource adapter.

The resource adapter provider must specify the following information in the deployment descriptor:

- **General information:** The resource adapter provider should specify the following general information about a resource adapter:
  - Name of the resource adapter
  - Description of the resource adapter
  - URI of a UI icon for the resource adapter
  - Name of the vendor who provides the resource adapter
  - Licensing requirement and description. Note that the management of licensing is outside the scope of the connector architecture.
  - Type of the EIS system supported - example: name of a specific database, ERP system, mainframe TP system without any versioning information
  - Version of the connector architecture specification (represented as a string) supported by the resource adapter
  - Version of the resource adapter represented as a string
- **ManagedConnectionFactory class**: The resource adapter provider must specify the fully-qualified name of the Java class that implements the `javax.resource.spi.ManagedConnectionFactory` interface.
- **ConnectionFactory interface and implementation class**: The resource adapter provider must specify the fully-qualified name of the Java interface and implementation class for the connection factory.
- **Connection interface and implementation class**: The resource adapter provider must specify the fully-qualified name of the Java interface and implementation class for the connection interface.
- **Transactional Support**: The resource adapter provider must specify the level of transaction support provided by the resource adapter implementation. The level of transaction support is required to be any one of the following: `NoTransaction`, `LocalTransaction` or `XATransaction`. Note that this support is specified for a resource adapter and not for the underlying EIS instance.
  - `NoTransaction`: The resource adapter does not support either resource manager local or JTA transactions. It does not implement either `XAResource` or `LocalTransaction` interfaces.

- `LocalTransaction`: The resource adapter supports resource manager local transactions by implementing the `LocalTransaction` interface. The local transaction management contract is specified in the section 6.7.
- `XATransaction`: The resource adapter supports both resource manager local and JTA transactions by implementing the `LocalTransaction` and `XAResource` interfaces respectively. The requirements for supporting `XAResource` based contract are specified in the section 6.6.

- **Configurable properties per ManagedConnectionFactory instance**: The resource adapter provider specifies name, type, description and an optional default value for the properties that have to be configured on a per `ManagedConnectionFactory` instance.

  Each `ManagedConnectionFactory` instance creates connections to a specific EIS instance based on the properties configured on the `ManagedConnectionFactory` instance. The configurable properties are specified only once in the deployment descriptor, even though a resource adapter can be used to configure multiple `ManagedConnnectionFactory` instances (that create connections to different instances of the same underlying EIS type).
- **Authentication Mechanism**: The resource adapter provider must specify all authentication mechanisms supported by the resource adapter. This includes the support provided by the resource adapter implementation but not by the underlying EIS instance. The standard values are: `BasicPassword` and `Kerbv5`. A resource adapter may support one or more of these authentication mechanisms.
  - `BasicPassword`: user-password based authentication mechanism that is specific to an EIS.
  - `Kerbv5`: Kerberos version 5 based authentication mechanism.

  If no authentication mechanism is specified as part of the deployment descriptor, the resource adapter supports no standard security authentication mechanism as part of the security contract.
- **Reauthentication support**: The resource adapter provider must specify whether a resource adapter supports re-authentication of an existing physical connection.
- **Extended Security Permissions**: The security permissions listed in the deployment descriptor are different from those required by the default permission set (refer to Section 11.2 for more details on security permissions).

The deployment descriptor specified by the resource adapter provider for its resource adapter must be consistent with the `XML DTD` specified in Section 10.6.

**Note**: The connector architecture does not specify standard deployment properties for the configuration of non-Java parts (example: native libraries) of a resource adapter. This applies only to the properties of the non-Java part not exposed through the Java part of the resource adapter. The non-Java part of a resource adapter should be configured using mechanisms specific to a resource adapter.

### 10.3.2 Deployer

The deployer is responsible for using a deployment tool to configure a resource adapter in a target operational environment. The configuration of a resource adapter is based on the properties defined in the deployment descriptor as part of the resource adapter module.

#### Stand-alone Resource Adapter Module

The deployment tool must first read the `ra.xml` deployment descriptor from the resource adapter module `.rar` file.

The deployment tool must extract all pieces of the resource adapter module— native libraries, Java classes— and install these pieces in the application server. The configuration of the

individual pieces is based on the deployment requirements specified in the deployment descriptor.

#### Resource Adapter Module with J2EE Application

Refer J2EE platform specification [8] for the requirements specified for the deployment of a J2EE application.

#### Configuration

The deployer must perform the following tasks to configure a resource adapter:

- Configure one or more property sets (one property set per `ManagedConnectionFactory` instance) for creating connections to various underlying EIS instances. The deployer creates a property set by using a deployment tool to set valid values for various configurable fields. The configuration of each field is based on the name, type and description of the field specified in the deployment descriptor.

  Each property set represents a specific configuration (to be set on a `ManagedConnectionFactory` instance) for creating connections to a specific EIS instance. Since a resource adapter may be used to create connections to multiple instances of the same EIS, there can be multiple property sets (one for each configured `ManagedConnectionFactory` instance) for a single resource adapter. For example, a deployment tool may create multiple copies of XML based deployment descriptor; each copy of the deployment descriptor carries a specific configuration of properties for a `ManagedConnectionFactory`.
- Configure application server mechanisms for the transaction management based on the level of transaction support specified by the resource adapter.
- Configure security in the target operational environment based on the security requirements specified by the resource adapter in its deployment descriptor.

#### Security Configuration

The security configuration is based on the following:

- whether or not the resource adapter supports a specific authentication mechanism and credentials interface. The deployment descriptor includes an element `authentication-mechanism` that specifies a supported authentication mechanism and the corresponding credentials interface.
- whether or not the application server is configured to support a specific mechanism type. For example, if the application server is not configured for Kerberos mechanism, then it is not capable of passing Kerberos credentials to the resource adapter as part of the security contract.

During the deployment, the deployer may (though is not required to) check whether or not an underlying EIS supports the same capabilities (example: transaction support, authentication mechanisms) as the corresponding resource adapter.

For example, if a resource adapter provides implementation support for Kerberos based authentication but the underlying EIS instance does not support Kerberos, then deployer may decide not to configure Kerberos for authentication to this EIS instance. However if the deployer does not perform such checks during deployment, any invalid configurations should lead to runtime exceptions.

### 10.3.3 Application Server

An application server provides a deployment tool that supports deployment of multiple resource adapters. Third-party enterprise tools vendors can also provide deployment tools. The connector architecture does not specify how third-party tools integrate with an application server. This is beyond the scope of the connector architecture.

A deployment tool must be capable of reading the deployment descriptor from a resource adapter module. It must enable the deployer to configure a resource adapter in the operational environment and thereby reflect the values of all properties declared in the deployment descriptor for the resource adapter.

A deployment tool must support management of multiple property sets (one per configured `ManagedConnectionFactory` instance) for a resource adapter. This includes support for adding or removing a property set from the configuration for a resource adapter.

A deployment tool must support the addition and removal of resource adapters from an operational environment.

An application server should use the deployment properties that describe the capabilities of a resource adapter (for example, support level for transactions) to provide different QoS for a configured resource adapter and its underlying EIS.

## 10.4 Interfaces/Classes

This section specifies Java classes / interfaces related to the configuration of a resource adapter in an operational environment.

### 10.4.1 ManagedConnectionFactory

The class that implements `ManagedConnectionFactory` interface supports a set of properties. These properties provide information required by the `ManagedConnectionfactory` for the creation of physical connections to the underlying EIS.

A resource adapter can implement the `ManagedConnectionFactory` interface as a Java Bean. As a Java Bean implementor, the resource adapter can also provide a `BeanInfo` class that implements the `java.beans.BeanInfo` interface and provides explicit information about the methods and properties supported by the `ManagedConnectionFactory` implementation class.

The implementation of `ManagedConnectionFactory` as a Java Bean improves the ability of tools (for tools that are based on the JavaBeans framework) to manage the configuration of `ManagedConnectionFactory` instances.

**Note**: The deployment tool is required to use an XML-based deployment descriptor (refer to Section 10.6) to determine the set of configurable properties for a `ManagedConnectionFactory`. A deployment descriptor for a resource adapter specifies the name, type, description, and default value of the configurable properties.

### 10.4.2 Properties Conventions

The `ManagedConnectionFactory` implementation class must provide getter and setter methods for each of its supported properties. The supported properties must be consistent with the specification of configurable properties specified in the deployment descriptor.

The getter and setter methods convention must be based on the Java Beans design pattern. These methods are defined on the implementation class and not on the `ManagedConnectionFactory` interface. This requirement keeps the `ManagedConnectionfactory` interface independent of any resource adapter or EIS-specific properties.

### 10.4.3 Standard Properties

The connector architecture identifies a standard set of properties common across various types of resource adapters and EISs. A resource adapter is not required to support a standard property if that property does not apply to its configuration.

These standard properties are defined as follows:

- `ServerName` name of the server for the EIS instance
- `PortNumber` port number for establishing a connection to an EIS instance

- `UserName` name of the user establishing a connection to an EIS instance
- `Password` password for the user establishing a connection
- `ConnectionURL` URL for the EIS instance to which to connect.

In addition to these standard properties, a `ManagedConnectionFactory` implementation class may support properties specific to a resource adapter and its underlying EIS.

All properties are administered by the deployer and are not visible to an application component provider.

The specified properties are required to be implemented as either bound or constrained properties. Refer Java Beans specification for details on bound and constrained properties.

In the XML deployment descriptor, any bounds or well-defined values of properties should be described in the `description` element.

## 10.5 JNDI Configuration and Lookup

This section specifies requirements for the configuration of the JNDI environment for a resource adapter.

In both managed and non-managed application scenarios, an application component (or application client) is required to look up a connection factory instance in the component's environment using the JNDI interface. The application component then uses the connection factory instance to get a connection to the underlying EIS. Section 5.4 specifies the application programming model in more detail.

The following code extract shows the JNDI lookup of a `javax.resource.cci.ConnectionFactory` instance.

```
// Application Component/Client Code
obtain the initial JNDI context
Context initctx = new InitialContext();

// perform JNDI lookup to obtain connection factory
javax.resource.cci.ConnectionFactory cxf =
     (javax.resource.cci.ConnectionFactory)initctx.lookup(“java:comp/
env/eis/MyEIS”);

javax.resource.cci.Connection cx = cxf.getConnection();
```

### 10.5.1 Responsibilities

In both managed and non-managed environments, registration of a connection factory instance in the JNDI namespace must use either the JNDI `Reference` or `Serializable` mechanism.

The choice between the two JNDI mechanisms depends on:

- Whether or not the JNDI provider being used supports a specific mechanism.
- Whether or not the application server and resource adapter provide the necessary support (specified in the respective requirements).
- Constraints on the size of serialized objects that can be stored in the JNDI namespace. The reference mechanism allows for only a reference to the actual object to be stored in the JNDI namespace. This is preferable to the serializable mechanism, which stores the whole serialized object in the namespace.

The following section specifies responsibilities of the roles involved in the JNDI configuration of a resource adapter.

### Deployer

The deployer is responsible for configuring connection factory instances in the JNDI environment. The deployer should manage the JNDI namespace such that the same programming model (as shown in section 10.5) for the JNDI-based connection factory lookup is supported in both managed and non-managed environments.

### Resource Adapter

The implementation class for a connection factory interface is required to implement both `java.io.Serializable` and `javax.resource.Referenceable` interfaces to support JNDI registration.

The following code extract shows the `javax.resource.Referenceable` interface:

```
public interface javax.resource.Referenceable
      extends javax.naming.Referenceable {

      public void setReference(Reference ref);
}
```

The `ManagedConnectionFactory` implementation class is required to implement the `java.io.Serializable` interface.

To support `Reference` mechanism in a non-managed environment, a resource adapter or a helper class is required to provide an implementation of the `javax.naming.spi.ObjectFactory` interface.

### Application Server

The implementation class for `javax.resource.spi.ConnectionManager` is required to implement the `java.io.Serializable` interface.

An application server is required to provide an implementation class for the `javax.naming.spi.ObjectFactory` interface to support JNDI `Reference` mechanism-based connection factory lookup. The implementation of this interface is application server specific.

Section 10.5.3 specifies more details on the `Reference` mechanism-based JNDI configuration in a managed environment.

## 10.5.2 Scenario: Serializable

The implementation classes for both `javax.resource.cci.ConnectionFactory` and `javax.resource.spi.ManagedConnectionFactory` interfaces implement the `java.io.Serializable` interface.

The deployment code retrieves the configuration properties from the XML deployment descriptor for the resource adapter. The deployment code then creates an instance of the `ManagedConnectionFactory` implementation class and configures properties on the instance.

```
// Deployment Code
// Create an instance of ManagedConnectionFactory implementation class
com.myeis.ManagedConnectionFactoryImpl mcf =
      new com.myeis.ManagedConnectionFactoryImpl();

// Set properties on ManagedConnectionFactory instance
// Note: Properties are defined on the implementation class and not on the
// javax.resource.spi.ManagedConnectionFactory interface
mcf.setServerName("...");
mcf.setPortNumber("...");
...
```

Note that in a non-managed environment, an application developer writes the deployment code. In a managed environment, the deployment tool typically hides the deployment code.

The deployment code uses the `ManagedConnectionFactory` instance to create a connection factory instance. The code then registers the connection factory instance in the JNDI namespace.

```
// Deployment Code
// In managed environment, create a ConnectionManager specific to
// the application server. Note that in a non-managed environment,
// ConnectionManager will be specific to the resource adapter.
com.wombatserver.ConnectionManager cm =
            new com.wombatserver.ConnectionManager(...);

// Create an instance of connection factory
Object cxf = mcf.createConnectionFactory(cm);

// Get JNDI context
javax.naming.Context ctx = new javax.naming.InitialContext(env);

// Bind to the JNDI namespace specifying a factory name
ctx.bind("...", cxf);
```

When an application component does a JNDI lookup of a connection factory instance, the returned connection factory instance should get associated with a configured `ManagedConnectionFactory` instance and a `ConnectionManager` instance. The implementation class for connection factory should achieve the association between these instances in an implementation-specific manner.

The following section illustrates JNDI configuration in a managed environment based on the `Reference` mechanism. This section uses the CCI interfaces `javax.resource.cci.ConnectionFactory` and `javax.resource.cci.Connection` as connection factory and connection interfaces respectively.

### 10.5.3 Scenario: Referenceable

The implementation class for the `ConnectionFactory` interface implements the `javax.resource.Referenceable` shown in the following code extract. Refer to the JNDI specification for more details on the `Referenceable` interface.

```
public class com.myeis.ConnectionFactoryImpl implements
                  javax.resource.Referenceable,
                  java.io.Serializable,
                  javax.resource.cci.ConnectionFactory {

     // Reference to this ConnectionFactory
     javax.naming.Reference reference;

     // setReference is called by deployment code
     public void setReference(Reference ref) {
            reference = ref;
     }

     // getReference is called by JNDI provider during Context.bind
     public Reference getReference() throws NamingException {
            return reference;
     }
     ...
```

```
}
```

### ObjectFactory Implementation

An application server provides a class (in an application server-specific implementation) that implements the `javax.naming.spi.ObjectFactory` interface. Refer to the JNDI specification for more details on the `ObjectFactory` interface.

In the `ObjectFactory.getObjectInstance` method, the information carried by the `Reference` parameter (set in the `ConnectionFactoryImpl.setReference` method) is used to look-up the property set to be configured on the target `ManagedConnectionFactory` instance.

The mapping from a `Reference` instance to multiple configured property sets enables an application server to configure multiple `ManagedConnectionFactory` instances with respective property sets. An application server maintains the property set configuration in an implementation-specific way based on the deployment descriptor specification.

The implementation and structure of `Reference` is specific to an application server. The following code extract is an illustrative example. It illustrates an implementation of the `ObjectFactory.getObjectInstance` method:

```
public class com.wombatserver.ApplicationServerJNDIHandler
            implements javax.naming.spi.ObjectFactory {
     // ...
     public Object getObjectInstance(Object obj, Name name,
                 Context ctx, Hashtable env)
                throws Exception {

           javax.naming.Reference ref = (javax.naming.Reference)obj;

           // Using the information carried by the Reference instance,
           // (<referenceName, logicalName> in this example) lookup
           // a configured property set and then configure a
           // ManagedConnectionFactory instance with specified
           // properties.
           // ... [implementation specific]
           //
           // For example: the instantiation of ManagedConnectionFactory
           // implementation class and invocation of its setter method
           // can be done using Java Reflection mechanism.

           javax.resource.spi.ManagedConnectionFactory mcf = //...

           // create a Connection Manager instance specific to the
           // application server
           com.wombatserver.ConnectionManager cxManager = // ...

           // create a connection factory instance.
           // The ConnectionManager instance provided by the application
           // server gets associated with the created
           // connection factory instance
           return mcf.createConnectionFactory(cxManager);
     }
     ...
}
```

### Deployment

The following deployment code shows registration of a reference to a connection factory instance in the JNDI namespace:

```
// Deployment Code
javax.naming.Context ctx = new javax.naming.InitialContext(env);

// Create an instance of connection factory
com.myeis.ConnectionFactoryImpl cf =
            new com.myeis.ConnectionFactoryImpl();

// Create a reference for the ConnectionFactory instance
javax.naming.Reference ref = new javax.naming.Reference(
            ConnectionFactoryImpl.class.getName(),
            new javax.naming.StringRefAddr(
                  “<referenceName>”, “<logicalName>”),
            ApplicationServerJNDIHandler.class.getName(),
            null);

cf.setReference(ref);

// bind to the JNDI namespace specifying a name for the connection factory
ctx.bind("...", cf);
```

Note that the deployment code should be designed as generic (though the above example does not show it that way). The code should dynamically create an instance of a connection factory, create a `Reference` instance, and then set the reference.

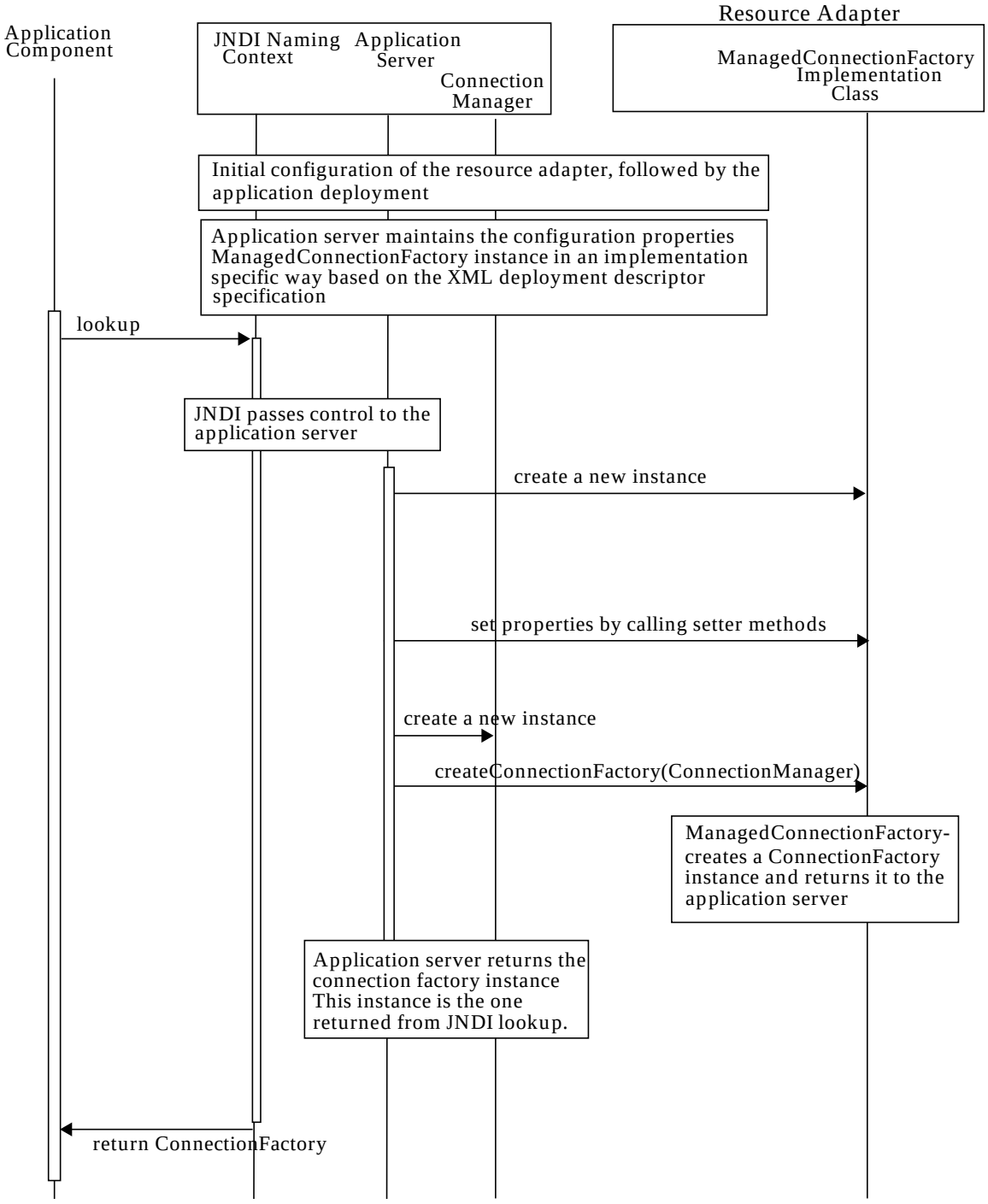
The `Context.bind` method registers a `Reference` to the connection factory instance in the JNDI namespace.

### Scenario: Connection Factory Lookup

The following steps occur when an application component calls the method JNDI `Context.lookup` to lookup a connection factory instance:

- JNDI passes control to the application server; the method `ObjectFactory.getObjectInstance` implemented by the application server is called.
- The application server creates a new instance of the `ManagedConnectionFactory` implementation class provided by the resource adapter.
- The application server calls setter methods on the `ManagedConnectionFactory` instance to set various configuration properties on this instance. These properties provide information required by the `ManagedConnectionFactory` instance to create physical connections to the underlying EIS. The application server uses an existing property set (configured during the deployment of a resource adapter) to set the required properties on the `ManagedConnectionFactory` instance.
- After the newly created `ManagedConnectionFactory` instance has been configured with its property set, the application server creates a new `ConnectionManager` instance.
- The application server calls the method `createConnectionFactory` on the `ManagedConnectionFactory` instance (passing in the `ConnectionManager` instance from the previous step) to get a `ConnectionFactory` instance.
- The application server returns the connection factory instance to the JNDI provider, so that this instance can be returned as a result of the JNDI lookup. The application component gets the `ConnectionFactory` instance as a result of the JNDI lookup.

**FIGURE 39.0** OID: Lookup of Connection Factory instance from JNDI



## 10.6 Resource Adapter XML DTD

This section specifies the XML DTD for the deployment descriptor for a resource adapter. The comments in the DTD specify additional requirements for syntax and semantics that cannot be specified by the DTD mechanism.

A resource adapter (or an application server on behalf of a resource adapter) may specify additional deployment information beyond the standard deployment descriptor. The additional information should be stored in a separate file and should refer to the standard deployment descriptor.

A resource adapter is not allowed to add any non-standard information into a standard deployment descriptor.

```
<!--
This is the XML DTD for the Connector 1.0 deployment descriptor.
All Connector 1.0 deployment descriptors must include a DOCTYPE of the follow-
ing form:

<!DOCTYPE connector PUBLIC"-//Sun Microsystems, Inc.//DTD Connector 1.0//EN"
"http://java.sun.com/dtd/connector_1_0.dtd">

-->

<!--
The following conventions apply to all J2EE deployment descriptor
elements unless indicated otherwise.

- In elements that contain PCDATA, leading and trailing whitespace
  in the data may be ignored.

- In elements whose value is an "enumerated type", the value is case sensitive.

- In elements that specify a pathname to a file within the same
  JAR file, relative filenames (i.e., those not starting with "/")
  are considered relative to the root of the JAR file's namespace.
  Absolute filenames (i.e., those starting with "/") also specify
  names in the root of the JAR file's namespace.  In general, relative
  names are preferred.  The exception is .war files where absolute
  names are preferred for consistency with the Servlet API.
-->


<!--
The connector element is the root element of the deployment descriptor
for the resource adapter. This element includes general information - vendor
name, version, specification version supported, icon -  about the
resource adapter module. It also includes information specific to the
implementation of the resource adapter library as specified through
the element resourceadapter.
-->
<!ELEMENT connector (display-name?, description?, icon?, vendor-name,
spec-version, eis-type, version, license?, resourceadapter)>

<!--
The element authentication-mechanism specifies an authentication mechanism
supported by the resource adapter. Note that this support is for
the resource adapter and not for the underlying EIS instance. The
optional description specifies any resource adapter specific requirement
```

```
for the support of security contract and authentication mechanism.

Note that BasicPassword mechanism type should support the
javax.resource.spi.security.PasswordCredential interface. The Kerbv5
mechanism type should support the javax.resource.spi.security.Generic-
Credential interface.

Used in: resourceadapter
-->
<!ELEMENT authentication-mechanism (
description?, authentication-mechanism-type, credential-interface)>

<!--
The element authentication-mechanism-type specifies type of an authentication
mechanism.

The example values are:
<authentication-mechanism-type>BasicPassword
                  </authentication-mechanism-type>
   <authentication-mechanism-type>Kerbv5
                  </authentication-mechanism-type>

Any additional security mechanisms are outside the scope of the
Connector architecture specification.

Used in: authentication-mechanism
-->
<!ELEMENT authentication-mechanism-type (#PCDATA)>

<!--
The element config-property contains a declaration of a single
configuration property for a ManagedConnectionFactory instance.

Each ManagedConnectionFactory instance creates connections to a
specific EIS instance based on the properties configured on the
ManagedConnectionFactory instance. The configurable properties are
specified only once in the deployment descriptor, even though a
resource adapter can be used to configure multiple ManagedConnnection-
Factory instances (that create connections to different instances of
the same EIS).

The declaration consists of an optional description, name, type
and an optional value of the configuration property. If the resource
adapter provider does not specify a value than the deployer is
responsible for providing a valid value for a configuration property.

Any bounds or well-defined values of properties should be described
in the description element.

Used in: resourceadapter
-->
<!ELEMENT config-property (description?, config-property-name,
config-property-type, config-property-value?)>

<!--
The element config-property-name contains the name of a configuration
property.

The connector architecture defines a set of well-defined properties
```

```
all of type java.lang.String. These are as follows:
     <config-property-name>ServerName</config-property-name>
     <config-property-name>PortNumber</config-property-name>
     <config-property-name>UserName</config-property-name>
     <config-property-name>Password</config-property-name>
     <config-property-name>ConnectionURL</config-property-name>

A resource adapter provider can extend this property set to include
properties specific to the resource adapter and its underlying EIS.

Used in: config-property

Example: <config-property-name>ServerName</config-property-name>
-->
<!ELEMENT config-property-name (#PCDATA)>

<!--
The element config-property-type contains the fully qualified Java
type of a configuration property as required by ManagedConnection-
Factory instance.

The following are the legal values of config-property-type:
   java.lang.Boolean, java.lang.String, java.lang.Integer,
   java.lang.Double, java.lang.Byte, java.lang.Short,
java.lang.Long, java.lang.Float, java.lang.Character

Used in: config-property

Example: <config-property-type>java.lang.String</config-property-type>
-->
<!ELEMENT config-property-type (#PCDATA)>


<!--
The element config-property-value contains the value of a configuration
entry.

Used in: config-property

Example: <config-property-value>WombatServer</config-property-value>
-->
<!ELEMENT config-property-value (#PCDATA)>

<!--
The element connection-impl-class specifies the fully-qualified
name of the Connection class that implements resource adapter
specific Connection interface.

Used in: resourceadapter

Example: <connection-impl-class>com.wombat.ConnectionImpl
         </connection-impl-class>
-->
<!ELEMENT connection-impl-class (#PCDATA)>

<!--
The element connection-interface specifies the fully-qualified
name of the Connection interface supported by the resource
adapter.
```

```
Used in: resourceadapter

Example: <connection-interface>javax.resource.cci.Connection
         </connection-interface>
-->
<!ELEMENT connection-interface (#PCDATA)>

<!--
The element connectionfactory-impl-class specifies the fully-qualified
name of the ConnectionFactory class that implements resource adapter
specific ConnectionFactory interface.

Used in: resourceadapter

Example: <connectionfactory-impl-class>com.wombat.ConnectionFactoryImpl
         </connectionfactory-impl-class>
-->
<!ELEMENT connectionfactory-impl-class (#PCDATA)>

<!--
The element connectionfactory-interface specifies the fully-qualified
name of the ConnectionFactory interface supported by the resource
adapter.

Used in: resourceadapter

Example: <connectionfactory-interface>com.wombat.ConnectionFactory
         </connectionfactory-interface>
OR
<connectionfactory-interface>javax.resource.cci.ConnectionFactory
         </connectionfactory-interface>
-->
<!ELEMENT connectionfactory-interface (#PCDATA)>

<!--
The element credential-interface specifies the interface that the
resource adapter implementation supports for the representation
of the credentials. This element should be used by application server
to find out the Credential interface it should use as part of the
security contract.

The possible values are:
   <credential-interface>javax.resource.spi.security.PasswordCredential
        </credential-interface>
   <credential-interface>javax.resource.spi.security.GenericCredential
        </credential-interface>

Used in: authentication-mechanism
-->
<!ELEMENT credential-interface (#PCDATA)>

<!--
The description element is used to provide text describing the parent
element.  The description element should include any information that
the component file producer wants to provide to the consumer of the
component file (i.e., to the Deployer). Typically, the tools used by
the component file consumer will display the description when
processing the parent element that contains the description.
```

```
Used in: authentication-mechanism, config-property, connector, license,
security-permission
-->
<!ELEMENT description (#PCDATA)>

<!--
The display-name element contains a short name that is intended to be
displayed by tools.  The display name need not be unique.

Used in: connector

Example:

<display-name>Employee Self Service</display-name>
-->
<!ELEMENT display-name (#PCDATA)>

<!--
The element eis-type contains information about the type of the
EIS. For example, the type of an EIS can be product name of EIS
independent of any version info.

This helps in identifying EIS instances that can be used with
this resource adapter.

Used in: connector
-->
<!ELEMENT eis-type (#PCDATA)>

<!--
The icon element contains a small icon and large icon element
which specify the URIs for a small and a large GIF or JPEG icon
image to represent the application in GUI.

Used in: connector
-->
<!ELEMENT icon (small-icon?, large-icon?)>

<!--
The large-icon element contains the name of a file containing a large
(32 x 32) icon image. The file name is a relative path within the
component’s jar file.

The image may be either in the JPEG or GIF format.
The icon can be used by tools.

Used in: icon

Example:

<large-icon>employee-service-icon32x32.jpg</large-icon>
-->
<!ELEMENT large-icon (#PCDATA)>

<!--
The element license specifies licensing requirements for the resource
adapter module. This element specifies whether a license is required
to deploy and use this resource adapter, and an optional description
```

```
of the licensing terms (examples: duration of license, number of
connection restrictions).

Used in: connector
-->
<!ELEMENT license (description?, license-required)>

<!--
The element license-required specifies whether a license is required
to deploy and use the resource adapter. This element must be one of
the following:

     <license-required>true</license-required>
     <license-required>false</license-required>

Used in: license
-->
<!ELEMENT license-required (#PCDATA)>

<!--
The element managedconnectionfactory-class specifies the fully qualified
name of the Java class that implements the javax.resource.spi.Managed-
ConnectionFactory interface. This Java class is provided as part of
resource adapter's implementation of connector architecture specified
contracts.

Used in: resourceadapter

Example:
  <managedconnectionfactory-class>com.wombat.ManagedConnectionFactoryImpl
  </managedconnectionfactory-class>
-->
<!ELEMENT managedconnectionfactory-class (#PCDATA)>

<!--
The element reauthentication-support specifies whether the resource
adapter implementation supports re-authentication of existing Managed-
Connection instance. Note that this information is for the resource
adapter implementation and not for the underlying EIS instance.

This element must be one of the following:
        <reauthentication-support>true</reauthentication-support>
        <reauthentication-support>false</reauthentication-support>
Used in: resourceadapter
-->
<!ELEMENT reauthentication-support (#PCDATA)>

<!--
The element resourceadapter specifies information about the resource
adapter. The information includes fully-qualified names of
class/interfaces required as part of the connector architecture
specified contracts, level of transaction support provided,
configurable properties for ManagedConnectionFactory instances,
one or more authentication mechanisms supported and additional
required security permissions.

If there is no authentication-mechanism specified as part of
resource adapter element then the resource adapter does not
support any standard security authentication mechanisms as part
```

```
of security contract. The application server ignores the security
part of the system contracts in this case.

Used in: connector
-->

<!ELEMENT resourceadapter (
managedconnectionfactory-class, connectionfactory-interface,
connectionfactory-impl-class, connection-interface,
connection-impl-class, transaction-support, config-property*,
authentication-mechanism*, reauthentication-support, security-permission*
)>

<!--
The element security permission specifies a security permission that
is required by the resource adapter code.

The security permission listed in the deployment descriptor are ones
that are different from those required by the default permission set
as specified in the connector specification. The optional description
can mention specific reason that resource adapter requires a given
security permission.

Used in: resourceadapter
-->
<!ELEMENT security-permission (description?, security-permission-spec)>

<!--
The element permission-spec specifies a security permission based
on the Security policy file syntax. Refer the following URL for
Sun's implementation of security permission specification:

http://java.sun.com/products/jdk/1.3/docs/guide/security/PolicyFiles.ht-
ml#FileSyntax

Used in: security-permission
-->
<!ELEMENT security-permission-spec (#PCDATA)>

<!--
The small-icon element contains the name of a file containing a small
(16 x 16) icon image. The file name is a relative path within the
component's jar file.

The image may be either in the JPEG or GIF format.
The icon can be used by tools.

Used in: icon

Example:

<small-icon>employee-service-icon16x16.jpg</small-icon>
-->
<!ELEMENT small-icon (#PCDATA)>

<!--
The element spec-version specifies the version of the connector
architecture specification that is supported by this resource
adapter. This information enables deployer to configure the resource
```

```
adapter to support deployment and runtime requirements of the
corresponding connector architecture specification.

Used in: connector

Example:
  <spec-version>1.0</spec-version>
-->
<!ELEMENT spec-version (#PCDATA)>

<!--
The transaction-support element specifies the level of transaction
support provided by the resource adapter.
The value of transaction-support must be one of the following:
<transaction-support>NoTransaction</transaction-support>
 <transaction-support>LocalTransaction</transaction-support>
 <transaction-support>XATransaction</transaction-support>

Used in: resourceadapter
-->
<!ELEMENT transaction-support (#PCDATA)>

<!--
The element vendor-name specifies the name of resource adapter provider
vendor.

Used in: connector

Example:
  <vendor-name>Wombat Corp.</vendor-name>
-->
<!ELEMENT vendor-name (#PCDATA)>

<!--
The element version specifies a string-based version of the
resource adapter from the resource adapter provider.

Used in: connector

Example:
  <version>1.0</version>
-->
<!ELEMENT version (#PCDATA)>

<!--
The ID mechanism is to allow tools that produce additional deployment
information (i.e., information beyond the standard deployment
descriptor information) to store the non-standard information in a
separate file, and easily refer from these tool-specific files to the
information in the standard deployment descriptor.

Tools are not allowed to add the non-standard information into the
standard deployment descriptor.
-->

<!ATTLIST authentication-mechanism id ID #IMPLIED>
<!ATTLIST authentication-mechanism-type id ID #IMPLIED>
<!ATTLIST config-property id ID #IMPLIED>
<!ATTLIST config-property-name id ID #IMPLIED>
```

```
<!ATTLIST config-property-type id ID #IMPLIED>
<!ATTLIST config-property-value id ID #IMPLIED>
<!ATTLIST connection-impl-class id ID #IMPLIED>
<!ATTLIST connection-interface id ID #IMPLIED>
<!ATTLIST connectionfactory-impl-class id ID #IMPLIED>
<!ATTLIST connectionfactory-interface id ID #IMPLIED>
<!ATTLIST connector id ID #IMPLIED>
<!ATTLIST credential-interface id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST eis-type id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST license id ID #IMPLIED>
<!ATTLIST license-required id ID #IMPLIED>
<!ATTLIST managedconnectionfactory-class id ID #IMPLIED>
<!ATTLIST reauthentication-support id ID #IMPLIED>
<!ATTLIST resourceadapter id ID #IMPLIED>
<!ATTLIST security-permission id ID #IMPLIED>
<!ATTLIST security-permission-spec id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST spec-version id ID #IMPLIED>
<!ATTLIST transaction-support id ID #IMPLIED>
<!ATTLIST vendor-name id ID #IMPLIED>
<!ATTLIST version id ID #IMPLIED>
```

# 11 Runtime Environment

This chapter focuses on the Java portion of a resource adapter that executes within a Java compatible runtime environment. A Java runtime environment is provided by an application server (and its containers).

The chapter specifies the Java APIs that a J2EE-compliant application server (and its containers) must make available to a resource adapter at runtime. A portable resource adapter can rely on these APIs to be available on all J2EE-compliant application servers.

The chapter also specifies programming restrictions imposed on a resource adapter. These restrictions enable an application server to enforce security and manage a runtime environment with multiple configured resource adapters.

## 11.1 Programming APIs

A resource adapter provider relies on a J2EE compliant application server to provide the following APIs:

- Java 2 SDK, Standard Edition, version 1.3 that includes the following as part of either the core platform or standard extensions: JavaIDL, JNDI Standard Extension, RMI-IIOP.
- Required APIs for Java 2 SDK, Enterprise Edition, version 1.3 as specified in the J2EE platform specification [8], version 1.3.
- JAAS 1.0 that requires Java 2 SDK, Standard Edition, version 1.3 or the Java 2 Runtime Environment version 1.3.

## 11.2 Security Permissions

An application server must provide a set of security permissions for execution of a resource adapter in a managed runtime environment. A resource adapter must be granted explicit permissions to access system resources.

Since the exact set of required security permissions for a resource adapter depends on the overall security policy for an operational environment and the implementation requirements of a resource adapter, the connector architecture does not define a fixed set of permissions.

The following permission set represents the default set of security permissions that a resource adapter should expect from an application server. These security permissions are described in detail in the Java 2 platform documentation. Refer document `http://java.sun.com/products/jdk/1.3/docs/guide/security/permissions.html`.

**Table 2: Default Security Permission Set**

| Security Permission | Default Policy | Notes |
|---|---|---|
| `java.security.AllPermission` | deny | Extreme care should be taken before granting this permission to a resource adapter. This permission should only be granted if the resource adapter code is completely trusted and when it is prohibitively cumbersome to add necessary permissions to the security policy. |
| `java.awt.AWTPermission` | deny * | A resource adapter must not use AWT code to interact with display or input devices. |
| `java.io.FilePermission` | grant `read` and `write <pathname>`<br><br>deny rest | A `java.io.FilePermission` represents access to a file or directory. A `FilePermission` consists of a pathname and a set of actions valid for that pathname.<br>A resource adapter is granted permission to read / write files as specified by the `pathname`, which is specific to a configured operational environment.<br>It is important to consider the implications of granting `Write` permission for `<<ALL FILES>>` because this grants the resource adapter permissions to write to the entire file system. This can allow a malicious resource adapter to mangle system binaries for the JVM environment. |
| `java.net.NetPermission` | deny * | |
| `java.util.PropertyPermission` | grant `read`<br>(allows `System.getProperty` to be called)<br><br>deny rest | Granting code permission to access certain system properties (`java.home`) can potentially give malevolent code sensitive information about the system environment (the Java installation directory). |
| `java.lang.reflect.ReflectPermission` | deny * | |

**Table 2: Default Security Permission Set**

| Security Permission | Default Policy | Notes |
|---|---|---|
| `java.lang.RuntimePermission` | deny * | By default, all `RuntimePermission` are denied to the resource adapter code.<br>A resource adapter should explicitly request `LoadLibrary.{libraryName}` to link a dynamic library. The `libraryName` represents a specific library.<br>A resource adapter that manages thread must explicitly request permission to `modifyThread` through its deployment descriptor.<br>A resource adapter should never be granted `exitVM` permission in the managed application server environment. |
| `java.security.SecurityPermission` | deny * | |
| `java.net.SocketPermission` | grant `connect` *<br><br>deny rest | A `java.net.SocketPermission` represents access to a network via sockets. A `SocketPermission` consists of a host specification and a set of actions specifying ways to connect to that host.<br>A resource adapter is granted permission to connect to any host as indicated by the wildcard *; |
| `java.security.SerializablePermission` | deny * | This ensures that a resource adapter cannot subclass `ObjectOutputStream` or `ObjectInputStream` to override the default serialization or deserialization of objects or to substitute one object for another during serialization or deserialization. |

## 11.3 Requirements

A resource adapter provider must ensure that resource adapter code does not conflict with the default security permission set. By ensuring this, a resource adapter can be deployed and run in any application server without execution or manageability problems.

If a resource adapter needs security permissions other than those specified in the default set, it is required to describe such requirements in the XML deployment descriptor using the `security-permission` element.

A deployment descriptor-based specification of an extended permission set for a resource adapter allows the deployer to analyze the security implications of the extended permission set and make a deployment decision accordingly. An application server must be capable of deploying a resource adapter with the default permission set.

**Example**

The resource adapter implementation creates a `java.net.Socket` and retrieves the hostname using the method `getHostName` on `java.net.InetAddress`.

**Table 3: Methods and Security Permissions required**

| Method | Security Manager Method Called | Permission |
|---|---|---|
| `java.net.Socket Socket(...)` | checkConnect({host}, {port}) | java.net.SocketPermission "{host}:{port}", "connect" |
| `java.net.InetAddress public String getHostName()` | checkConnect({host}, -1) | java.net.SocketPermission "{host}", "resolve" |
| ... | ... | ... |

The default `SocketPermission` (as specified in Table 2) is `grant connect` and `deny rest`. This means that if resource adapter uses the default permission set, the first method `Socket(...)` will be allowed while the second method `InetAddress.getHostName` is disallowed.

The resource adapter needs to explicitly request security permission for `InetAddress.getHostName` method in the `security-permission-spec` element of its XML deployment descriptor. The following is an example of specification of additional security permission:

```
<security-permission-spec>
     grant  {
            permission java.net.SocketPermission *, "resolve";
            };
</security-permission-spec>
```

## 11.4 Privileged Code

The Java 2 security architecture requires that whenever a system resource access or any secured action is attempted, all code traversed by the current execution thread up to that point must have the necessary permissions for the system resource access, unless some code on the thread has been marked as privileged. Refer to `http://java.sun.com/products/jdk/1.3/docs/guide/security/doprivileged.html`.

A resource adapter runs in its own protection domain as identified by its code source and security permission set. For the resource adapter to be allowed to perform a secured action (example, writing a file), it must have granted permission for that particular action.

Resource adapter code is considered system code which may require more security permissions than the calling application component code. For example, when an application component calls a resource adapter method to execute a function call on the underlying EIS instance, the resource adapter code may need more security permissions (example: the ability to create a thread) than allowed to the calling component.

To support such scenarios, the resource adapter code should use the `privileged` code feature in the Java security architecture. This enables the resource adapter code to temporarily perform more secured actions than are available directly to the application code calling the resource adapter.

# 12 Exceptions

This chapter specifies standard exceptions that identify error conditions which may occur as part of the connector architecture.

The connector architecture defines two classes of exceptions:

- `System Exceptions`—Indicate an unexpected error condition that occurs as part of an invocation of a method defined in the system contracts. For example, system exceptions are used to indicate transaction management-related errors. A system exception is targeted for handling by an application server or resource adapter (depending on who threw the exception), and may not be reported in its original form directly to an application component.
- `Application Exceptions`—Thrown when an application component accesses an EIS resource. For example, an application exception might indicate an error in the execution of a function on a target EIS. These exceptions are meant to be handled directly by an application component.

The connector architecture defines the class `javax.resource.ResourceException` as the root of the system exception hierarchy. The `ResourceException` class extends the `java.lang.Exception` class and is a checked exception.

The `javax.resource.ResourceException` is also the root of the application exception hierarchy for CCI. Application level exceptions are specified in more details in Java docs for CCI.

## 12.1 ResourceException

A `ResourceException` provides the following information:

- A resource adapter-specific string describing the error. This string is a standard Java exception message and is available through the `getMessage` method.
- A resource adapter-specific error code that identifies the error condition represented by the `ResourceException`.
- A reference to another exception. Often a `ResourceException` results from a lower-level problem. If appropriate, a lower-level exception (a `java.lang.Exception` or any derived exception type) may be linked to a `ResourceException` instance.

## 12.2 System Exceptions

The connector architecture requires that methods (as part of a system contract implementation) use checked `ResourceException` (and other standard exceptions derived from it) to indicate system-level error conditions. Using checked exceptions leads to a strict enforcement of the contract for throwing and catching of system exceptions and dealing with error conditions.

In addition, a method implementation may use `java.lang.RuntimeException` or any derived exception to indicate runtime error conditions of varying severity levels. Using unchecked exceptions to indicate important system-level error conditions is not recommended for an implementation of system contracts.

If a method needs to indicate a serious error condition that it does not want the caller to catch, the method should use `java.lang.Error` to indicate such conditions. A method is not required to declare in its throws clause any subclasses of `Error` that may be thrown but not caught during the execution of the method, since these errors are abnormal conditions that should never occur.

### Exception Hierarchy

The `ResourceException` represents a *generic* form of exception. A derived exception represents a *specific class* of error conditions. This design enables the method invocation code to catch a *class* of error conditions based on the exception type and to handle error conditions appropriately.

The following exceptions are derived from `ResourceException` to indicate more *specific classes* of system error conditions:

- `javax.resource.spi.SecurityException`: A `SecurityException` indicates error conditions related to the security contract between an application server and resource adapter. The common error conditions represented by this exception are:
    - Invalid security information (represented by a `Subject` instance) passed across the security contract. For example, credentials may have an expired or an invalid format.
    - Lack of support for a specific security mechanism in an EIS or resource adapter.
    - Failure to create a connection to an EIS because of failed authentication or authorization.
    - Failure to authenticate a resource principal to an EIS or failure to establish a secure association with an underlying EIS instance.
    - Access control exception indicating that a requested access to an EIS resource or a request to create a new connection has been denied.
- `javax.resource.spi.LocalTransactionException`: A `LocalTransactionException` represents various error conditions related to the local transaction management contract. The JTA specification specifies the `javax.transaction.xa.XAException` class for exceptions related to a `XAResource`-based transaction management contract. The `LocalTransactionException` is used for the local transaction management contract to indicate the following types of error conditions:
    - Invalid transaction context when a transaction operation is executed. For example, calling `LocalTransaction.commit` method without an active local transaction is an error condition.
    - Transaction is rolled back instead of being committed in the `LocalTransaction.commit` method.
    - Attempt to start a local transaction from the same thread on a `ManagedConnection` instance that is already associated with an active local transaction.
    - All resource adapter or resource manager-specific error conditions related to local transaction management. Examples are violation of integrity constraints, deadlock detection, communication failure during transaction completion, or any retry requirement.
- `javax.resource.spi.ResourceAdapterInternalException`: This exception indicates all system-level error conditions related to a resource adapter. The common error conditions indicated by this exception type are:
    - Invalid configuration of the `ManagadConnectionFactory` for creation of a new physical connection. An example is an invalid server name for a target EIS instance.
    - Failure to create a physical connection to a EIS instance due to a communication protocol error or a resource adapter implementation-specific error.
    - Error conditions internal to a resource adapter implementation.

- `javax.resource.spi.EISSystemException`: An `EISSystemException` is used to indicate any EIS specific system-level error condition. Examples of common error conditions are: failure or inactivity of an EIS instance, communication failure, and a EIS-specific error during the creation of a physical connection.
- `javax.resource.spi.ApplicationServerInternalException`: This exception is thrown by an application server to indicate error conditions specific to an application server. Example error conditions are: errors related to an application server configuration or implementation of mechanisms internal to an application server (example: connection pooling, thread management).
- `javax.resource.spi.ResourceAllocationException`: This exception is thrown by an application server or resource adapter to indicate a failure to allocate system resources (example: threads, physical connections). An example is an error condition that results when an upper bound is reached for the maximum number of physical connections that can be managed by an application server-specific connection pool.
- `javax.resource.spi.IllegalStateException`: This exception is thrown from a method if the invoked code (either the resource adapter or the application server for system contracts) is in an illegal or inappropriate state for the method invocation.
- `javax.resource.NotSupportedException`: This exception is thrown to indicate that an invoked code (either the resource adapter or the application server for system contracts) cannot execute an operation because the operation is not a supported feature. For example, if the transaction support level for a resource adapter is `NoTransaction`, an invocation of `ManagedConnection.getXAResource` method throws a `NotSupportedException` exception.
- `javax.resource.spi.CommException`: This exception indicates errors related to failed or interrupted communication with an EIS instance. Examples of common error conditions represented by this exception type include: communication protocol error, invalidated connection due to server failure.

## 12.3 Additional Exceptions

The JTA specification specifies the `javax.transaction.xa.XAException` class for exceptions related to `XAResource`-based transaction management contract.

# 13 Projected Items

**[Targeted for .next release cycle]**

The following set of features are strong candidates for inclusion in the future versions of the connector architecture:

- **Pluggable JMS providers**: This will involve the specification of a standard architecture for the pluggability of multiple JMS providers into an application server. This may require enhancement of the present system contracts (transaction, security, connection management) and addition of new system contracts specific to JMS pluggability.

  A standard architecture for the pluggability of JMS providers will be an important factor in driving the adoption of EJB-JMS integration (specified as part of the EJB 2.0 specification [1]).

- **Thread Management Contract**: This will involve an extension of the system contracts to support a thread management contract for asynchronous interactions with the underlying EIS.

- **Common Client Interface**: The CCI may become required as part of a future version of the connector architecture. The CCI may also be extended to include support for XML, type mapping and metadata facility.

# Appendix A: Caching Manager

The following section describes how the connector architecture supports caching.

This section serves as a brief introduction to the caching support in the connector architecture. A future version of the connector architecture will address this issue in detail. The caching manager architecture will be aligned with the related work being done as part of the EJB 2.0 specification.

## A.1 Overview

The connector architecture provides a standard way of extending an application manager for plugging in caching managers. A caching manager may be provided by a third party vendor or a resource adapter provider.

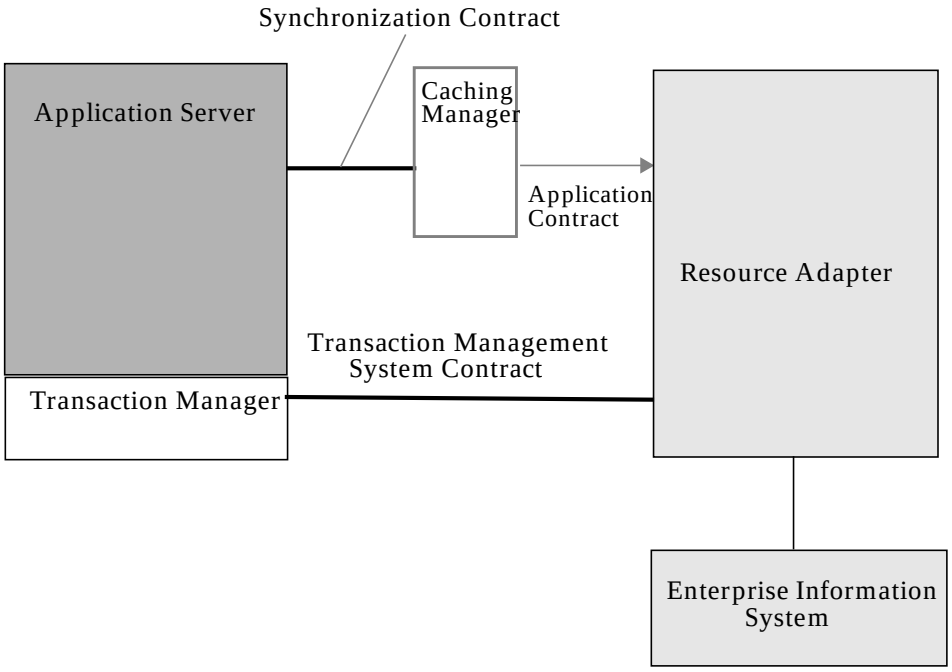
A caching manager manages cached state for application components while they access EISs across transactions.

A caching manager is provided above a resource adapter. An application component may access a resource manager either through a caching manager (thereby maintaining a cached state across application requests) or directly through the resource adapter with no caching involved.

The `XAResource` based transaction management contract enables an external transaction manager to control and coordinate transactions across multiple resource managers. A caching manager (provided above the resource adapter) needs to be synchronized relative to the transaction coordination flow (defined by the JTA `XAResource` interface) on the underlying resource manager. This leads to a requirement for a synchronization contract between the application server and caching manager.

The connector architecture defines a standard synchronization contract between the application server and caching manager. The caching manager uses the synchronization notifications to manage its cached state and to flush it to the resource adapter. The resource adapter then takes the responsibility of managing its recoverable units of work and participates in the transaction coordination protocol from the transaction manager.

**FIGURE 40.0** Synchronization Contract between Caching Manager and Application Server




The above diagram shows a caching manager layered above a resource adapter. The contract between caching manager and resource adapter is specific to a resource adapter.

## A.2 Synchronization contract

**Note:** To support a caching manager as a standard extension to the application server, additional contracts between the application server and the caching manager are required. This version of the specification introduces only the synchronization contract.

This section specifies the synchronization contract between the application server and the caching manager.

### Interface

Each caching manager implements the `javax.transaction.Synchronization` interface. A caching manager registers its `Synchronization` instance with the application server when it is configured with the application server.

The caching manager receives synchronization notifications only for transactions managed by an external transaction manager. In the case of transactions managed internally by a resource manager, the resource adapter and caching manager define their own implementation-specific mechanisms for synchronizing caches.

The `Synchronization.beforeCompletion` method is called prior to the start of the two-phase commit transaction completion process. This call executes in the same transaction context of the caller who initiated the transaction completion. The caching manager uses this notification to flush its cached state to the resource adapter.

The `Synchronization.afterCompletion` method is called after the transaction has completed. The status of transaction completion is passed in as a parameter. The caching manager uses this notification to do any cache cleanups if a rollback has occurred.

### Implementation

The caching manager is required to support the `javax.transaction.Synchronization` interface. If the caching manager implements the `Synchronization` interface and registers it with the application server, then the application server must invoke the `beforeCompletion` and `afterCompletion` notifications.

The application server is responsible for ensuring that synchronization notifications are delivered first to the application components (that have expressed interest in receiving synchronization notification through their respective application component and container-specific mechanisms) and then to the caching managers that implement the `Synchronization` interface.

# Appendix B: Security Scenarios

The following section describes various scenarios for EIS integration. These scenarios focus on security aspects of the connector architecture.

Note that these scenarios establish the requirements to be addressed by the connector architecture. The chapters 7 and 8 specify the requirements that are supported in the version 1.0 of the specification.

A J2EE application is a multi-tier, web-enabled application that accesses EISs. It consists of one or more application components—EJBs, JSPs, servlets—which are deployed on containers. These containers can be one of the following:

- Web containers that host JSP, servlets, and static HTML pages
- EJB containers that host EJB components
- Application client containers that host standalone application clients

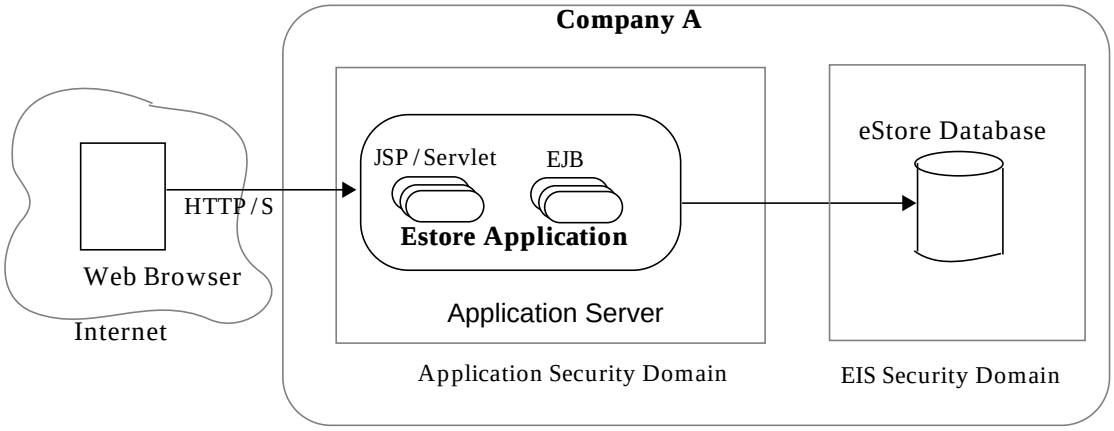
In the following scenarios, the description of the architecture and security environments are illustrative in scope.

## B.1 EStore Application

Company A has an eStore application based on the J2EE platform. The eStore application is composed of EJBs and JSP / servlets; together they collaborate to provide the overall functionality of the application. The application also utilizes an eStore database to store data related to product catalog;, shopping carts; customer registration and profiles; transaction status and records; and order status.

The architecture of this application is illustrated in the following diagram:

**FIGURE 41.0** Illustrative Architecture of an Estore Application




### Scenario

A customer, using a web browser, initiates an e-commerce transaction with the eStore application. The e-commerce transaction consists of a series of customer actions. The customer:

- Browses the catalog
- Makes a selection of products
- Puts the selected products into a shopping cart
- Enters her user name and password to initiate a secure transaction

- Fills in order-related information
- And, finally, places an order

In this scenario, the eStore application stores all persistent information about customers and their transactions in a database.

### Security Environment

To support the above interaction scenario, the system administrator configures a unique security domain (with specific security technology and security policies) for the eStore application. A firewall protects this security domain from unauthorized Internet access.

The security domain configuration for the eStore application includes secure web access to the eStore application. Secure web access is set up based on the requirements specified in the J2EE specification. Note that the focus of this section is security related to EIS integration, not on web access security. As a result, this description ignores web access security.

The system administrator sets up a database to manage persistent data for the eStore application. In terms of security, the database system is configured with an independent security domain. This domain has its own set of user accounts, plus its own security policies and mechanisms for authentication and authorization.

The system administrator (or database administrator DBA) creates a unique database account (called `EStoreUser`) to handle database transactions; the database transactions correspond to different customer-driven interactions with the eStore application. He also sets up an additional database account (called `EStoreAdministrator`) to manage the database on behalf of the eStore administrator. This administrative account has a higher level of access privileges.

To facilitate better scaling of the eStore application, the system administrator may choose to set the load balancing of database operations across multiple databases. He may also partition persistent data and transactions across multiple database accounts, based on various performance optimization criteria. These areas are out of the scope for this document.

This scenario deals only with the simple case of a single database and a single user account to handle all database transactions.

### Deployment

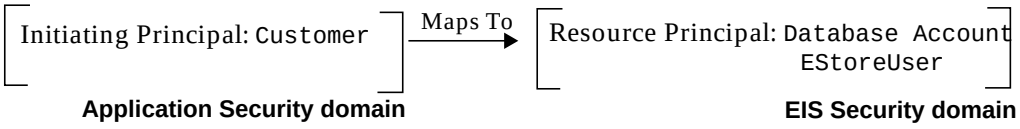
**Note**: This document does not address how principal delegation happens between the web and EJB containers. When an EJB instance acquires an EIS connection, a caller principal is associated with the EJB instance. This document does not address determining which caller principal is associated with the EJB instance.

During the deployment of the eStore application, the deployer sets up access control for all authenticated customer accounts—the customer accounts that are driving e-commerce transactions over the web—based on a single role `eStoreUserRole`.

The deployer configures the resource adapter with the security information that is required for the creation of database connections. This security information is the database account `EStoreUser` and its password.

The deployer sets up the resource principal for accessing the database system as illustrated in the Figure 42.0:

**FIGURE 42.0** Resource Principal for Estore Application Scenario




The deployment configuration ensures that all database access is always performed under the security context of the database account `EStoreUser`.

All authenticated customers (referred to as `Initiating Principal`) map to a single `EStoreUser` database account. The eStore application uses an implementation-specific mechanism to tie database transactions (performed under a single database account) to the unique identity (social security number or eStore account ID) of the initiating principal. To ensure that database access has been properly authorized, the eStore application also performs access control based on the role of the initiating principal. Because all initiating principals map to a single role, this is in effect a simple case.

This scenario describes an n-to-1 mapping. However, depending on the requirements of an application, the deployer can set the principal mapping to be different from an n-to-1 mapping. For example, the deployer can map each role to a single resource principal, where a role corresponds to an initiating principal. This results in a [*m* principals and *n* roles] to [*p* resource principals] mapping. When doing such principal mapping, the deployer has to ensure not to compromise the access rights of the mapped principals. An illustrative example is:

- User is in administrator role: Principal `EISadmin`
- User is in manager role: Principal `EISmanager`
- User is in employee role: Principal `EISemployee`

## B.2 Employee Self Service Application

Company B has developed and deployed an employee self-service (ESS) application based on the J2EE platform. This application supports a web interface to the existing Human Resources (HR) applications, which are supported by the ERP system from Vendor X. The ESS application also provides additional business processes customized to the needs of Company B.

The application tier is composed of EJBs and JSPs that provide the customization of the business processes and support a company-standardized web interface. The ESS application enables an employee (under the roles of Manager, HR manager, and Employee) to perform various HR functions, including personal information management, payroll management, compensation management, benefits administration, travel management, and HR cost planning.

### Architecture

The IS department of Company B has deployed its HR ESS application and ERP system in a secure environment on a single physical location. Any access to the HR application is permitted Only legal employees of the organization are permitted access to the HR application. Access is based on the employee's roles and access privileges. In addition, access to the application can only be from within the organization-wide intranet. See Figure 43.0.

### Security Environment

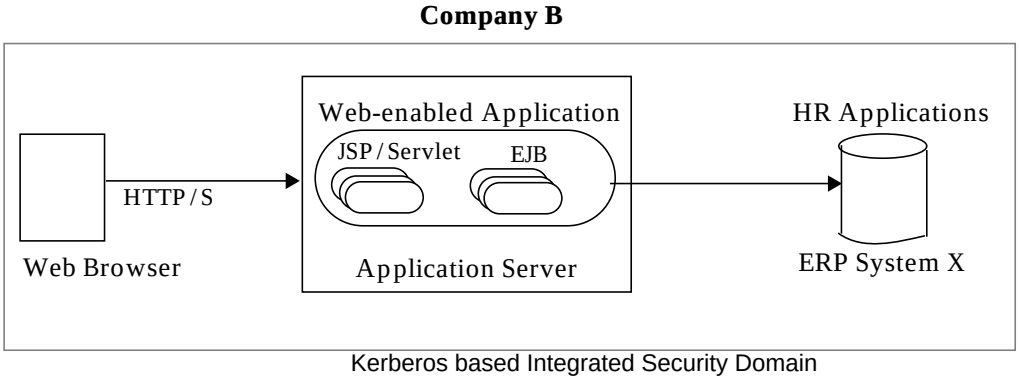
To support the various interaction scenarios related to the ESS application, the system administrator sets up an end-to-end Kerberos-based security domain for this application environment.

**Note**: The Security policies and mechanisms that are required to achieve this single security domain are technology dependent. Refer to Kerberos V5 specification for more details.

The system administrator configures the security environment to support single sign-on; the user logs on only once and can then access all the services provided by the ESS application and its underlying ERP system. Single sign-on is achieved through the security mechanism and policies specific to the underlying security technology, which in this case is Kerberos.

The ERP system administrator configures all legal employees as valid user accounts in the ERP system. He also must set up various roles (Manager, HRManager, and Employee), default passwords, and access privileges. This security information is kept synchronized with the enterprise-wide directory service, which is used by Kerberos to perform the initial authentication of end-users.

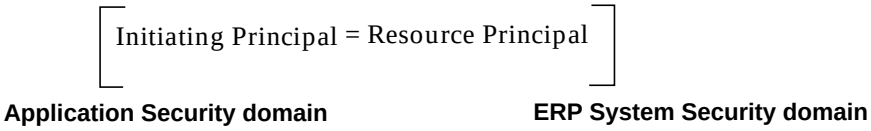
**FIGURE 43.0** Illustrative Architecture of an Employee Self-service Application




### Deployment

During deployment of the ESS application, the deployer sets a default delegation policy of client impersonation for EIS sign-on. In this case, the application server and ERP system know that it is the initiating principal accessing their respective services and they perform access control based on this knowledge. See Figure 44.0.

**FIGURE 44.0** Principal Mapping




In this scenario, both the initiating principal and the resource principal refer to the same principal. This common principal is authenticated using Kerberos and its Kerberos credentials are valid in the security domains of both the application and the ERP system.

The deployer sets up access control for all authenticated employees (initiating principal) based on the configured roles—Manager, HR Manager, and Employee.

If the ERP system does not support Kerberos, then an alternate scenario is utilized. The deployer or application server administrator sets up an automatic mapping of Kerberos credentials (for the initiating principal) to valid credentials (for the same principal) in the security domain of the ERP system. Note that when the ERP system does support Kerberos, the application server performs no credentials mapping.

### Scenario

An employee initiates an initial login to his client desktop. He enters his username and password. As part of this initial login, the employee (called initiating principal C) gets authenticated with Kerberos KDC. [Refer to the details for Kerberos KDC authentication in the Kerberos v5 specification.]

After a successful login, the employee starts using his desktop environment. He directs his web browser to the URL for the ESS application deployed on the application server. At this point, the initiating principal C authenticates itself to the application server and establishes a session key with the application server.

The ESS application is set up to impersonate initiating principal C when accessing the ERP system, which is running on another server. Though the application server directly connects to the ERP system, access to the ERP system is requested on behalf of the initiating principal. For this to work, principal C needs to delegate its identity and Kerberos credential to the application server and allow the application server to make requests to the ERP system on C's behalf.

## B.3 Integrated Purchasing Application

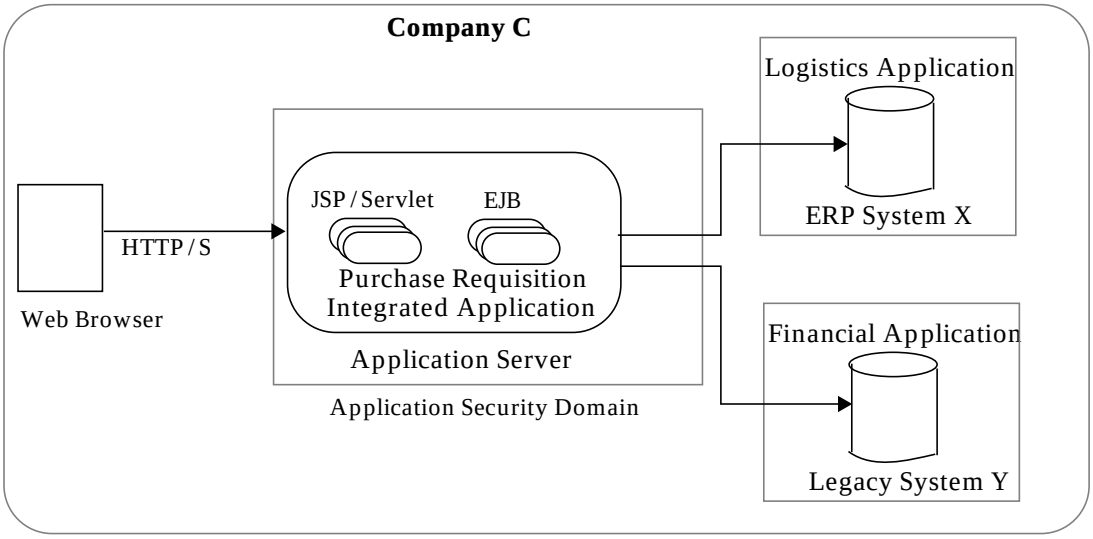
Company C has an integrated purchasing application that enables an employee to use a web-based interface to perform multiple purchasing transactions. An employee can manage the entire procurement process, from creating a purchase requisition through invoice approval. The purchasing application also integrates with the enterprise's existing financial applications so that the accounting and financial aspects of the procurement business processes can be tracked.

### Architecture

Figure 45.0 illustrates an architecture for this purchasing application. The application has been developed and deployed based on the J2EE platform and is composed of EJBs and JSPs. The EJB components provide the integration across the different applications—the logistics application from a separate vendor (this application provides integrated purchasing and inventory management functions) and the financial accounting applications (the applications supported by the legacy system from vendor Y).

Company B is a huge decentralized enterprise; its business units and departments are geographically distributed. In this scenario, different IS departments manage ERP system X and legacy system Y. In addition, ERP system X and legacy system Y have been deployed at secured data centers in different geographic locations. Lastly, the integrated purchasing application has been deployed at a geographic location different from both ERP system X and legacy system Y.

**FIGURE 45.0** Illustrative Architecture of an Integrated Purchasing Application




### Security Environment

ERP system X and legacy system Y are also in different security domains; they use different security technologies and have their own specific security policies and mechanisms. The integrated purchasing application is deployed in a security domain that is different from both that of ERP system X and legacy system Y.

To support the various interaction scenarios for this integrated purchasing application, the ERP system administrator creates a unique account `LogisticsAppUser` in the ERP system. He sets up the password and specific access rights for this account. This user account is allowed access only to the logistics business processes that are used by the integrated purchasing application.

Likewise, the system administrator for the legacy system creates a unique account `FinancialAppUser`. He also sets up the password and specific access rights for this account.

The application server administrator, as part of the operational environment of the application server, configures the access to an organization-wide directory. This directory contains security information (name, password, role, and access rights) for all the employees in the organization. It is used for authentication and authorization of employees accessing the purchasing application.

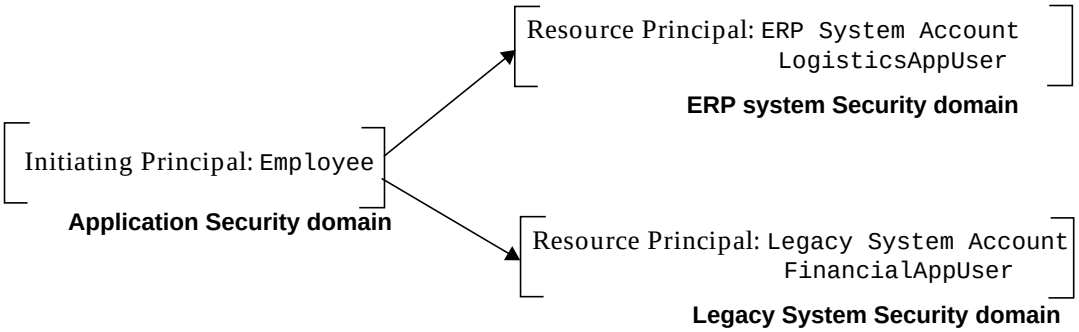
Due to their physical separation in this scenario, EISs X and Y are accessed over either a secure private network or over the Internet. This requires that a secure association be established between the application server and the EISs. A secure association allows a component on the application server to communicate securely with an EIS.

### Deployment

During the deployment of this application, the deployer configures the security information (that is, the user account `LogisticsAppUser` and its password) required to create connections to the ERP system. This configuration is done using the resource adapter for ERP system X. The deployer also configures the security information (that is, user account `FinancialAppUser` and its password) required to create connections to the legacy system Y.

The deployer configures security information in the application server to achieve the principal mapping shown in Figure 46.0.

**FIGURE 46.0** Principal Mapping




This principal mapping ensures that all connections to the ERP system are established under the security context of `LogisticsAppUser`, the resource principal for the ERP system security domain. Similarly, all connections to legacy system Y are established under the security context of the `FinancialAppUser`.

The application server does this principal mapping for all authenticated initiating principals (that is, employees accessing the integrated purchasing application) when the application connects to either the ERP system or the legacy system.

# Appendix C: JAAS based Security Architecture

This chapter extends the security architecture specified in Chapters 7 and 8 to include support for JAAS-based pluggable authentication. The chapter refers to the following documents:

- White Paper on User Authentication and Authorization in Java platform: `http://java.sun.com/security/jaas/doc/jaas.html`
- JAAS 1.0 documentation

## C.1 Java Authentication and Authorization Service (JAAS)

JAAS provides a standard Java framework and programming interface that enables applications to authenticate and enforce access controls upon users. JAAS is divided into two parts based on the security services that it provides:

- **Pluggable Authentication**: This part of the JAAS framework allows a system administrator to plug in the appropriate authentication services to meet the security requirements of an application environment. There is no need to modify or recompile an existing application to support new or different authentication services.
- **Authorization**: Once authentication has successfully completed, JAAS provides the ability to enforce access controls based upon the principals associated with an authenticated subject. The JAAS principal-based access controls (access controls based on who runs code) supplement the existing Java 2 code source-based access controls (access controls based on where code came from and who signed it).

## C.2 Requirements

The connector security architecture uses JAAS in two ways:

- **Security Contract**: The connector security architecture uses the JAAS `Subject` class as part of the security contract between an application server and a resource adapter. Use of JAAS interfaces enables the security contract to remain independent of specific security technologies or mechanisms. The security contract has been specified in Section 8.2.
- **JAAS Pluggable Authentication framework**: This framework lets an application server and its underlying authentication services remain independent from each other. When additional EISs and new authentication services are required (or are upgraded), they can be plugged in an application server without requiring modifications to the application server.

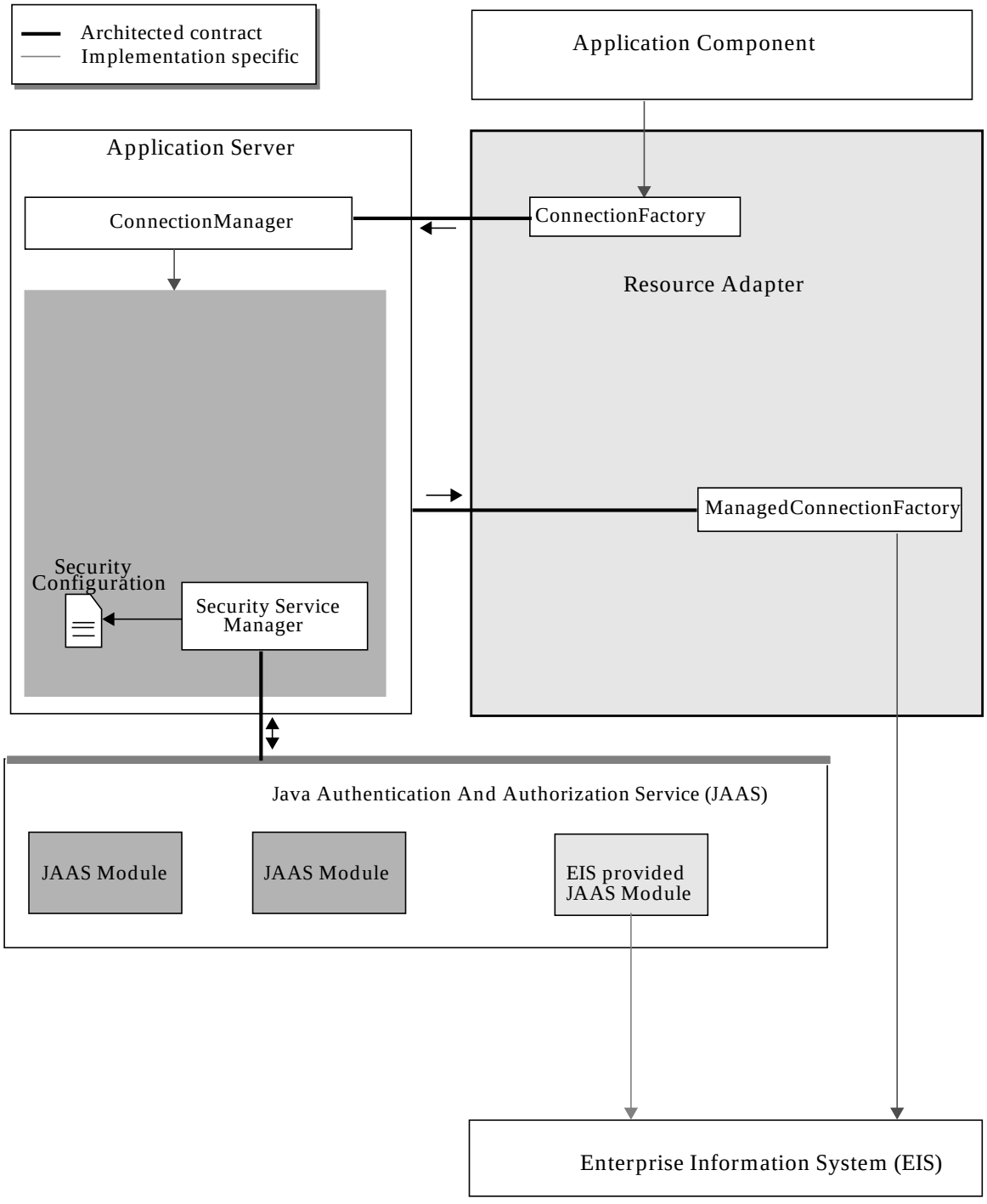
The connector architecture requires that the application server and the resource adapter must support the JAAS `Subject` class as part of the security contract. However, it recommends (but does not mandate) that an application server use the JAAS pluggable authentication framework.

The connector architecture does not require support for the authorization portion of the JAAS framework.

## C.3 Security Architecture

The following section specifies the JAAS based security architecture. The security architecture addresses how JAAS may be used by an application server to support authentication requirements of heterogeneous EISs.

**FIGURE 47.0** Security Architecture.




### C.3.1 JAAS Modules

The connector architecture recommends (but does not mandate) that an application server support *platform-wide* JAAS modules (also called authentication modules) for authentication mechanisms that are common across multiple EISs. The implementation of these JAAS modules is

typically specific to an application server. However, these modules may be developed to be reusable across application servers.

A resource adapter provider can provide a resource adapter-specific *custom* implementation of a JAAS module. The connector architecture recommends that a resource adapter provider provide a *custom* JAAS module when the underlying EIS supports an authentication mechanism that is EIS specific and is not supported by an application server.

A *custom* JAAS module can be packaged together with a resource adapter and can be pluggable into an application server using the JAAS architecture.

The JAAS specification [7] specifies requirements for developing and configuring JAAS modules.

### C.3.2 Illustrative Examples: JAAS Module

It is not a goal of the connector architecture to specify a standard architecture for JAAS modules. The following are illustrative examples of JAAS modules used typically in the JAAS-based security architecture:

#### Principal Mapping Module

The application server invokes the principal mapping module passing in the `Subject` instance corresponding to the caller / initiating principal. The JAAS specification specifies the interfaces / classes and mechanisms involved in the invocation of a JAAS module.

The principal mapping module maps a caller / initiating principal to a valid resource principal and returns the mapped resource principal as part of a `Subject` instance. The authentication data (example, password) for the mapped resource principal is added to the `Subject`'s credentials. The authentication data is used later to authenticate the resource principal to the underlying EIS.

A special case of the principal mapping module takes a `null` `Subject` as an input parameter and forms a `Subject` instance with a valid resource principal and authentication data. This is the case of default principal mapping.

The principal mapping module achieves its mapping functionality by using security information configured in the application server or an enterprise directory.

The principal mapping module does not authenticate a resource principal and is configured to perform only principal mapping. The authentication of a mapped resource principal is performed separately by an authentication mechanism-specific JAAS module.

#### Credential Mapping Module

The credential mapping module automatically maps credentials from one authentication domain to those in a different target authentication domain. For example, an application server can provide a module that maps the public key certificate-based credential associated with a principal to a Kerberos credential.

The credentials mapping module can use the JAAS callback mechanism (note that this involves no user-interface based interaction) to get authentication data from the application server. The authentication data is used to authenticate the principal to the target authentication domain during the credentials mapping. This module can also use an enterprise directory to get security information or pre-configured mapped credentials.

#### Kerberos Module

This type of JAAS module supports Kerberos-based authentication for a principal. A sample Kerberos module supports:

- Getting a TGT (ticket granting ticket) to the Kerberos server in the local domain. The TGT is created by the KDC. The TGT is placed on the credentials structure for a principal.

- Delegation of authentication based on either a forwardable or proxy mechanism as specified in the Kerberos specification.

## C.4 Generic Security Service API: GSS-API

The GSS-API is a standard API that provides security services to caller applications in a generic fashion. These security services include authentication, authorization, principal delegation, secure association establishment, per-message confidentiality, and integrity. These services can be supported by a wide range of security mechanisms and technologies. However, an application using GSS-API accesses these services in a generic mechanism-independent fashion and achieves source-level portability.

In the context of the connector architecture, a resource adapter uses GSS-API to establish a secure association with the underlying EIS. The use of the GSS mechanism by a resource adapter is typical in the following scenarios:

- The EIS supports Kerberos as a third-party authentication service and uses GSS-API as a generic API for accessing security services.
- The resource adapter and EIS need data integrity and confidentiality services during their communication over insecure links.

The GSS-API has been implemented over a range of security mechanisms, including Kerberos V5. There is presently a work in progress to provide a Java binding of GSS-API [6].

**Note**: The connector architecture does not require a resource adapter to use GSS-API.

## C.5 Security Configuration

During deployment of a resource adapter, the deployer is responsible for configuring JAAS modules in the operational environment. The configuration of JAAS modules is based on the security requirements specified by a resource adapter in its deployment descriptor. Refer to Section 10.6.

The element `authentication-mechanism` in the deployment descriptor specifies an authentication mechanism supported by a resource adapter. The standard types of authentication mechanisms are: `BasicPassword` and `Kerbv5`. For example, if a resource adapter specifies support for `kerbv5` authentication mechanism, the deployer configures a Kerberos JAAS module in the operational environment.

### JAAS Configuration

The deployer sets up the configuration of JAAS modules based on the JAAS-specified mechanism. Refer to `javax.security.auth.login.Configuration` specification for more details. The JAAS configuration includes the following information on a per resource adapter basis:

- One or more authentication modules used to authenticate a resource principal.
- The order in which authentication modules need to invoked during a stacked authentication.
- The flag value controlling authentication semantics if stacked modules are invoked.

The format for the above configuration is specific to an application server implementation.

## C.6 Scenarios

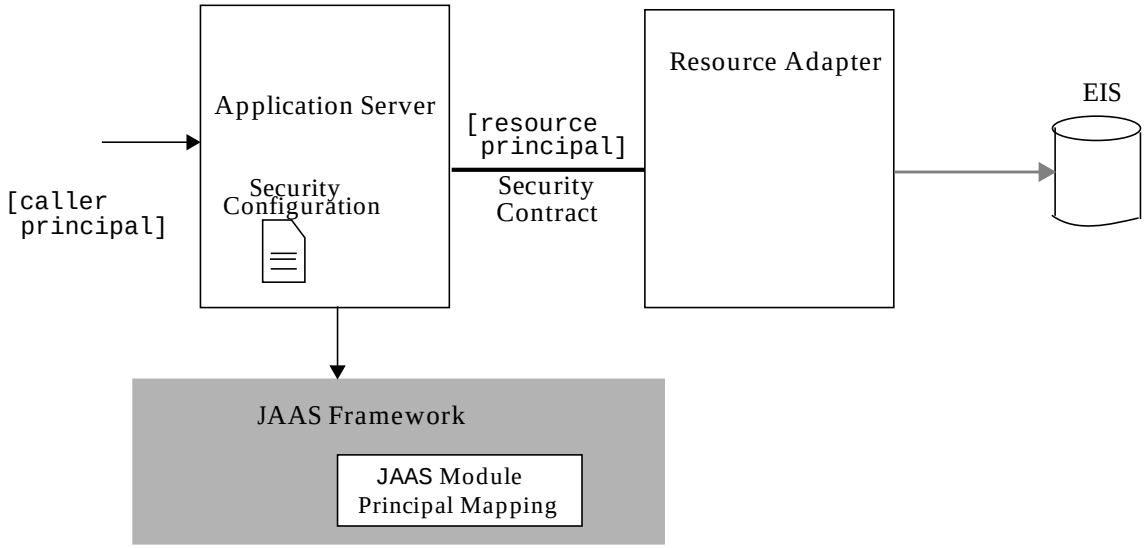
The following section illustrates security scenarios for JAAS based security architecture.

### C.6.1 Scenario: Resource Adapter Managed Authentication

This scenario enables the connector architecture to support EIS specific username and password-based authentication. It involves the following steps:

- The application component invokes connection request method on the resource adapter without passing in any security arguments. The resource adapter passes the connection request to the application server.
- During the deployment of the resource adapter, the application server is configured to use a principal mapping module. This principal mapping module takes a `Subject` instance with the caller principal and returns a `Subject` instance with a valid resource principal and `PasswordCredential` instance. The `PasswordCredential` has the password for authentication of the resource principal.
- The application server calls `LoginContext.login` method. On a successful return from the principal mapping module, the application server gets a `Subject` instance that has the mapped resource principal with a valid `PasswordCredential`.

**FIGURE 48.0** Resource Adapter-managed Authentication

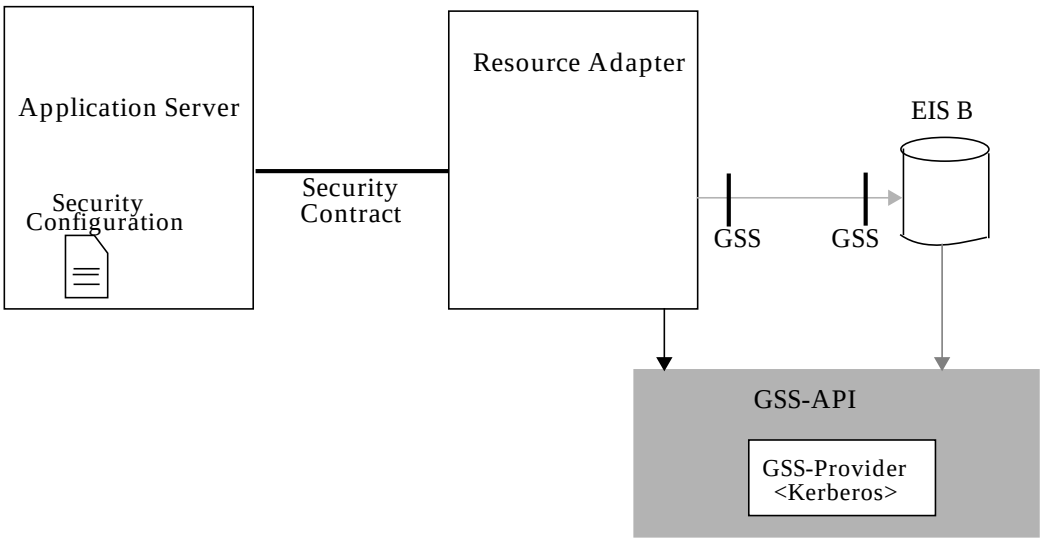



- The application server invokes the method `ManagedConnectionFactory.createManagedConnection` passing in a `non-null Subject` instance. The `Subject` instance carries the resource principal and its corresponding `PasswordCredential`, which holds the user name and password.
- The resource adapter extracts the user name and password from the `PasswordCredential` instance. The resource adapter uses the getter methods (`getPrivateCredentials` method) defined on the `Subject` interface to extract the `PasswordCredential` instance.
- The resource adapter uses username and password information (extracted from the `PasswordCredential` instance) to authenticate the resource principal to the EIS. The authentication happens during the creation of the connection through an authentication mechanism specific to the underlying EIS.

### C.6.2 Scenario: Kerberos and Principal Delegation

The scenario in Figure 49.0 involves the following steps:

**FIGURE 49.0** Kerberos Authentication with Principal Delegation




- The initiating principal has already authenticated itself to the application server using Kerberos. The initiating principal has a service ticket for the application server and a TGT (ticket granting ticket issued by the KDC) as part of its Kerberos based credentials.
- In this scenario, the application server is configured to impersonate the initiating principal when connecting to the EIS instance. So even though application server is directly connecting to the EIS, access to the EIS is being requested on behalf of the initiating principal. The initiating principal needs to pass its identity to the application server and allow the application server to make requests to the EIS on behalf of the initiating principal. The above is achieved through delegation of authentication.
- The application server calls the method `ManagedConnectionFactory.createManaged-Connection` by passing in a `Subject` instance with the initiating principal and its Kerberos credentials. The credentials contain a Kerberos TGT and are represented through the `GenericCredential` interface.
- The resource adapter extracts the resource principal and its Kerberos credentials from the `Subject` instance.
- The resource adapter creates a new physical connection to the EIS.
- If the resource adapter and EIS support GSS-API for establishing a secure association, the resource adapter uses the Kerberos credentials based on the GSS mechanism as follows. For details, see GSS-API specification:
    - resource adapter calls `GSS_Acquire_cred` method to acquire `cred_handle` in order to reference the credentials for establishing the shared security context.
    - resource adapter calls the `GSS_Init_sec_context` method. The method `GSS_Init_sec_context` yields a service ticket to the requested EIS service with the corresponding session key.

**Note**: The mechanism and representation through which Kerberos credentials are shared across the underlying JAAS module and GSS provider is beyond the scope of the connector architecture.

- After success, `GSS_Init_sec_context` builds a specific Kerberos-formatted message and returns it as an output token. The resource adapter sends the output token to the EIS instance.

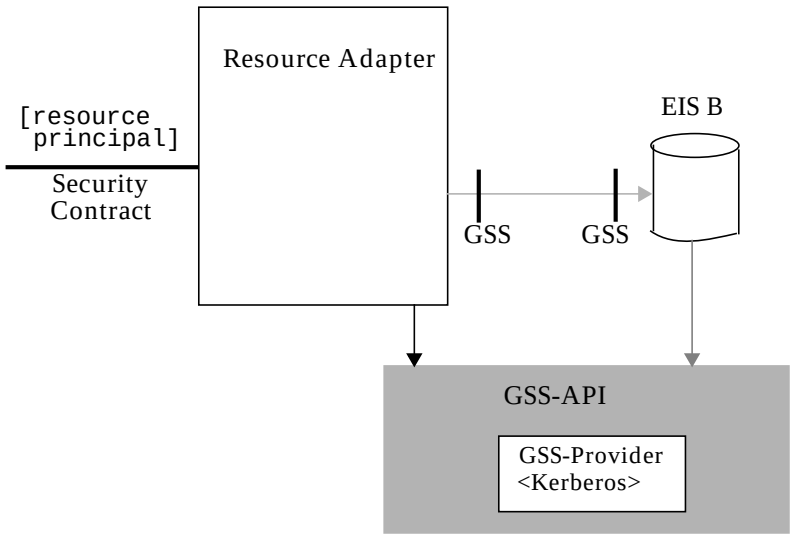
- EIS service passes the received token to the `GSS_Accept_sec_context` method.
- Resource adapter and EIS now hold the shared security context (so have established a secure association) in the form of a session key associated with the service ticket. They can now use the session key in the subsequent per-message methods: `GSS-GetMIC`, `GSS_VerifyMIC`, `GSS_Wrap`, `GSS_Unwrap`.

- If the resource adapter and EIS fail to establish a secure association, the resource adapter cannot use the physical connection as a valid connection to the EIS instance. The resource adapter returns a security exception on the `createManagedConnection` method.

### C.6.3 Scenario: GSS-API

If an EIS supports the GSS mechanism, a resource adapter may (but is not required to) use GSS-API to set up a secure association with the EIS instance. (See Figure 50.0.) The section Generic Security Service API: GSS-API on page 169 gives a brief overview of GSS-API.

**FIGURE 50.0** GSS-API use by Resource Adapter

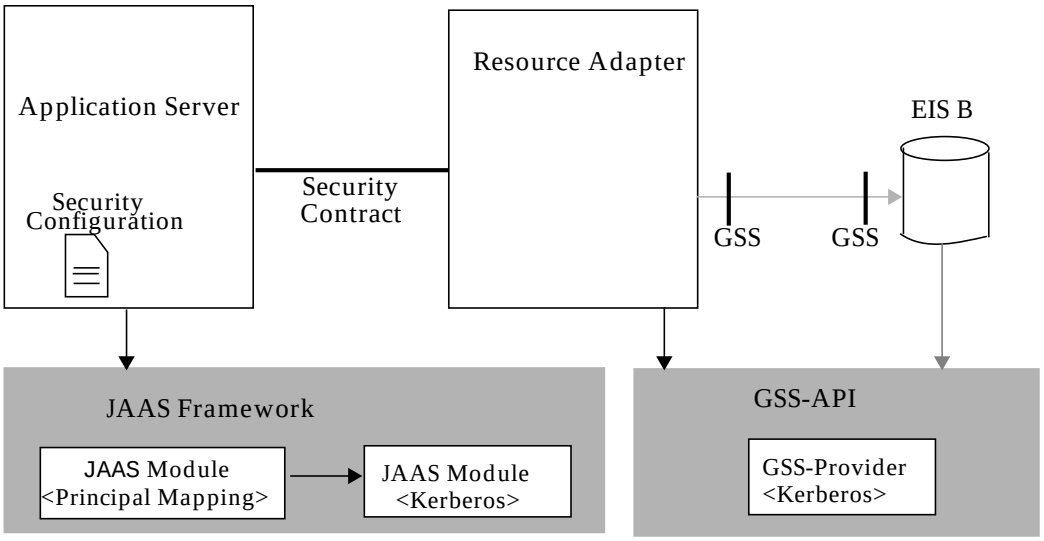



A formal specification of the use of GSS-API by a resource adapter is beyond the scope of the connector architecture. However, GSS-API has been mentioned as a possible implementation option for a resource adapter that has the GSS mechanism supported by its underlying EIS.

### C.6.4 Scenario: Kerberos Authentication after Principal Mapping

The scenario depicted in Figure 51.0 involves the following steps:

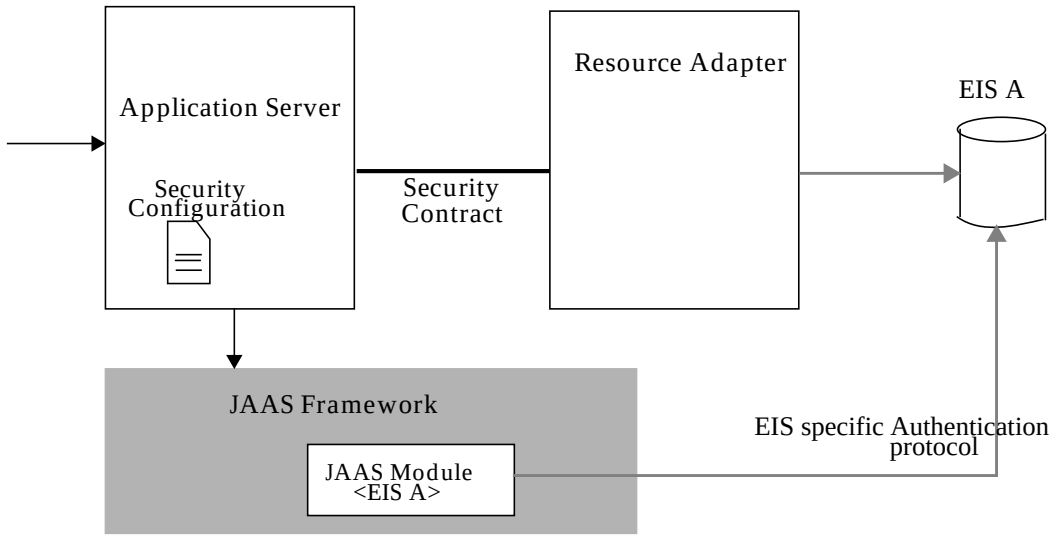
**FIGURE 51.0** Kerberos Authentication after Principal Mapping




- The application server is configured to use the principal mapping module and Kerberos module. The two authentication modules are stacked together with the principal mapping module first.
- The application server creates a `LoginContext` instance by passing in the `Subject` instance for the caller principal and a `CallbackHandler` instance. Next, the application server calls the `login` method on the `LoginContext` instance.
- The principal mapping module takes a `Subject` instance with caller principal and returns a `Subject` instance with a valid resource principal and Kerberos- based authentication data. The principal mapping module does not authenticate the resource principal; it does only principal mapping to find the mapped resource principal and its authentication data.
- Next, the Kerberos module (called after the principal mapping module) uses the resource principal and its authentication data to authenticate the resource principal. The Kerberos module leads to a valid TGT for the Kerberos domain supported by the EIS. The TGT is contained in the Kerberos credentials represented through the `GenericCredential` interface.
- The application server calls the method `ManagedConnectionFactory.createManagedConnection` passing in a `Subject` instance with the resource principal and its Kerberos credentials.
- The remaining steps are the same as in the previous scenario, Section C.6.2, “Scenario: Kerberos and Principal Delegation,” on page -170

### C.6.5 Scenario: EIS-specific Authentication

**FIGURE 52.0** Authentication though EIS specific JAAS Module




The scenario in Figure 52.0 involves the following steps:

- During the configuration of a resource adapter, the application server is configured to use an EIS-specific JAAS module for authentication to the underlying EIS.

  The configured JAAS module supports an authentication mechanism specific to the EIS. The application server is responsibility for managing the authentication data and JAAS configuration.

- The application server gets a request from the application component to create a new physical connection to the EIS. Creating a new physical connection requires the resource principal to authenticate itself to the underlying EIS instance.
- The application server initiates the authentication of the resource principal. It creates a `LoginContext` instance by passing in the `Subject` instance and a `CallbackHandler` instance. Next, the application server calls the `login` method on the `LoginContext` instance.
- The JAAS module authenticates the resource principal to the underlying EIS. It uses the callback handler provided by the application server to get the authentication data.
- The application server invokes the method `ManagedConnectionFactory. createManagedConnection` passing in the `Subject` instance with the authenticated resource principal and its credential.
- The resource adapter extracts the credential (associated with the `Subject` instance) for the resource principal using the getter methods defined on the `Subject` interface. The resource adapter uses this credential to create a connection to the underlying EIS.

In this scenario, authenticating a resource principal (initiated by the application server and performed by the JAAS module) is separate from creating a connection to the EIS. The resource adapter uses the credential of the resource principal to create a connection to the EIS. This connection creation can involve further authentication.

After successfully creating a connection to the EIS, the resource adapter returns the newly created connection from the method `ManagedConnectionFactory.createManagedConnection`.

# Appendix D: Related Documents

[1] Enterprise JavaBeans (EJB) specification, version 2.0:
`http://java.sun.com/products/ejb/`

[2] Java Transaction API (JTA) specification, version 1.x
`http://java.sun.com/products/jta/`

[3] JDBC API specification, version 3.0
`http://java.sun.com/products/jdbc/`

[4] X/Open CAE Specification -- Distributed Transaction Processing: the XA specification, X/Open document

[5] RFC: Generic Security Service API (GSS-API) specification, version 2:
`http://www.ietf.org/rfc/rfc2078.txt`

[6] Java Specification Request: Generic Security Service API (GSS-API), Java bindings:
`http://java.sun.com/aboutJava/communityprocess/jsr/jsr_072_gss.html`

[7] Java Authentication and Authorization Service, version 1.0:
`http://java.sun.com/security/jaas/`

[8] Java 2 Platform Enterprise Edition (J2EE), Platform specification, versions 1. 3:
`http://java.sun.com/j2ee/`

[9] Java Server Pages (JSP) specification, version 1.2:
`http://java.sun.com/products/jsp/`

[10] Servlets specification, version 2.3:
`http://java.sun.com/products/servlet/`

# Appendix E: Change History

## E.1 Version 0.9

- Editorial run through the document
- Added section 1.4 on relationship between JDBC and Connector architecture
- Added scenario on B2B in the chapter 4
- Added java.io.Serializable to the code specification of interfaces that are required to support Serializable interface
- Added clarifications in the chapter 5 based on the expert comments. The changes are marked by change bars.
- Added equals and hashCode methods to interface ConnectionRequestInfo
- Added section 6.8 on Connection Association
- Added clarifications to the chapter 7. Did minor restructuring of the chapter based on review comments. The changes are marked by change bars.
- Added clarifications to the chapter 8 based on expert comments
- Changed few details and added clarifications in the chapter 9 based on the review comments. The changes are marked by change bars.
- Added more description for packaging and deployment in the chapter 10
- Clarified version dependencies in the chapter 11
- Introduced interface javax.resource.Referenceable for the standard setReference method
- Removed scenarios on Credentials Mapping and Single sign-on from Appendix C. Updated scenario C.6.2 to refer GSS-API.

## E.2 Version 1.0 - Public Draft 1

- Removed definition of "Connector" from 2.1. The term Connector is now used broadly refer to the Connector architecture, while resource adapter refers to the system library.
- Added requirement for ConnectionEventListener to 6.9.2: Application Server
- Added connection handle property to the ConnectionEvent, section 5.5.7
- Introduced getResultSetInfo method in the Connection interface
- Added "Administered Object" in the section 9.6.2
- Added more details to section "Auto Commit" in 9.5.2
- Introduced separate interface for ResultSetInfo in the section 9.10.3
- Changed specification of element config-property-type in section 10.6
- Added an example to illustrate security permission specification in the section 11.3
- Added CCI related information to Projected Items, chapter 12

## E.3 Version 1.0 - Public Draft 2

- Section 5.5.1: Change based on introduction of ConnectionSpec interface
- Section 5.5.1: Added clarification to ConnectionRequestInfo section
- Section 5.5.4: Added clarification to section on "Cleanup of ManagedConnection"
- Section 5.5.6: Added clarification to paragraph after the interface for ConnectionEventListener
- Section 5.9.1: Added clarification to description of the scenario

- Section 6.8: Moved earlier section "Details on Local transaction" ahead of connection sharing section and renamed it "Scenarios: Local Transaction Management". No change in any content.
- Section 6.9: Added more details on connection sharing based on the changes in EJB 2.0 and J2EE 1.3 platform specification.
- Section 6.10: Added this section to clarify local transaction optimization. This is based on changes in EJB 2.0 and J2EE 1.3 platform specification.
- Section 6.11: Made a new section on "Scenarios: Connection sharing". No change in content.
- Section 6.12: Added clarifications and requirements in the section on "Connection Association"
- Section 6.13.2: Moved requirements on connection sharing to section 6.9
- Section 7.4.2: Code sample changed to reflect ConnectionSpec usage
- Section 9.5.1: Changed getConnection(Map) to getConnection(ConnectionSpec) and added clarifications.
- Section 9.5.2: Introduced a section on ConnectionSpec
- Section 9.7.2: Added methods to ResourceAdapterMetaData interface. Added description of these methods.
- Section 9.9.1: Record, MappedRecord and IndexedRecord now extend Serializable interface.
- Section 9.10: Added note on JDBC semantics in relation to CCI ResultSet
- Section 9.10.3: Added note on ResultSetInfo implementation requirements
- Section 10.6: Change to auth-mechanism specification in DTD. Removed + from credential-interface.
- Figure 29: Added clarifications for the diagram
- Section 8.3: Clarified security contract requirements for the application server
- Section 9.5.1: Moved method getRecordFactory from Interaction to ConnectionFactory. Note that it is not necessary to have an active connection to create generic record instances.

## E.4 Version 1.0 - Proposed Final Draft 2

- Reviewed requirements in terms of compliance testing. marked with change bars in the document
- Fixed documentation errors
- System Contracts:
  - Section 5.5.4: Clarified requirements for the method `matchManagedConnections` on `ManagedConnectionFactory` interface
  - Section 6.9: Made requirements for connection sharing consistent with J2EE 1.3 platform specification
  - Section 6.10: Added specification of requirements for different transaction scenarios. Added illustrative scenarios
  - Section 6.11: Removed a transaction scenario that illustrated connection sharing
  - Section 6.11: Clarified requirements for connection association
- Common Client Interface:
  - Removed `setLogWriter`, `getLogWriter`, `setTimeout`, `getTimeout` methods from `ConnectionFactory` interface
  - Added description for exceptions in Java docs for the CCI interfaces. Note that no new exception has been introduced

- `ConnectionFactory` implementation class required to provide a default constructor
- Added clarifications; marked by change bars

- Deployment and Packaging:
  - Section 10.2: Clarified requirements for packaging and deployment of a resource adapter
- DTD changes based on a review of DTDs for various J2EE specifications:
  - Ordered elements alphabetically except the root element
  - <display-name> changed to optional in <connector> element
  - Used common elements from other DTDs: <description>, <small-icon>, <large-icon>
  - Used common header comments across all J2EE DTDs
  - <auth-mechanism> changed to <authentication-mechanism>
  - <auth-mech-type> changed to <authentication-mechanism-type>
  - Added `java.lang.Character` to <config-property-type>:
  - Changed defined values in <authentication-mechanism-type>: `basic-password` to `BasicPassword`, `kerbv5` to `Kerbv5`
  - Changed defined values in <transaction-support> element: `no_transaction` to `NoTransaction`, `local_transaction` to `LocalTransaction`, `xa_transaction` to `XATransaction`

## E.5 Version 1.0 - Final Release

- Clarification on reauthentication in the section 8.2.7
- Change in auto-commit in section 9.5.3. Removed set / getAutoCommit methods from the Connection interface

JavaSoft
2550 Garcia Avenue
Mountain View, CA 94043
408-343-1400

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 9844 5000
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: +358-0-525561
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 5717-5000
Korea: 822-563-8700
Latin America: 415 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-849 2828
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
415 960-1300
Intercontinental Sales: 415 688-9000