Studiengang: Informatik

Betreuer: Dipl. Inf. Ralf Wagner
Prüfer: Prof. Dr.-Ing. B. Mitschang,

Beginn am: 15. Juli 2002
Beendet am: 14. Januar 2003

CR-Nummern: C.1.3, E.1, H.3.4, H.4.1

Diplomarbeit-Nr. 2038

Konzeption und Realisierung von Workflow-Adaptern in der J2EE-Architektur

Lifang Chen

Fakultät Informatik
Universität Stuttgart
Institut für Parallele und Verteilte Systeme
Abteilung Anwendersoftware
Breitwiesenstraße 20-22
D-70565 Stuttgart

Kurzfassung

Diese Diplomarbeit behandelt die Implementierung eines Resource-Adapters für das Workflow-Management-System MQSeries Workflow in der J2EE-Connector-Architektur.

Die J2EE-Connector-Architektur definiert die Interfaces, die ein Resource-Adapter und ein Applikationsserver implementieren muss. Zwischen einem Applikationsserver und einem Resource-Adapter definiert die J2EE-Connector-Architektur den System-Vertrag, zu dem das Transaction-Management, Security-Management und das Connection-Management gehören. Ein Resource-Adapter bietet auch das Common-Client-Interface für die Applikationskomponenten, durch das die Applikationskomponente mit dem Resource-Adapter und schließlich mit dem EIS kommunizieren kann.

In dieser Diplomarbeit ist der Resource-Adapter speziell auf MQSeries-Workflow ausgerichtet. Der Resource-Adapter ist dafür zuständig, eine Verbindung von dem Applikationsserver zu dem MQSeries-Workflow zu erzeugen und zu beenden, und die Kommunikation zwischen den Applikationskomponenten und dem MQSeries-Workflow zu führen.

Am Ende dieser Diplomarbeit wird das Zeitverhalten des Resource-Adapters untersucht. Die Zeitdauer für einen Verbindungsaufbau und für einen Verbindungsabbau werden für zwei Fälle gemessen und verglichen, nämlich dem Managed-Applikationsszenario und dem Non-Managed-Applikationsszenario. Außerdem wird das Zeitverhalten des Resource-Adapters in Abhängigkeit von Datenmengen untersucht.

Inhaltsverzeichnis

1 Einleitung 1

2 Java 2 Platform, Enterprise Edition(J2EE) 3

2.1 J2EE Architektur 3

2.2 Enterprise JavaBeans 6

3 J2EE Connector-Architektur 8

3.1 Overview 8

3.2 JDBC API und Common-Client-Interface 9

3.3 Connector-Architektur 9

3.3.1 System-Contracts 10

3.3.2 Common-Client-Interface(CCI) 11

3.3.3 Non-Managed-Applikationsszenario 12

4 Workflow Management System(WFMS) 13

4.1 Grundbegriffe 13

4.2 Standardisierte Workflow-Management-System 14

4.3 MQ Series Workflow(MQWF) 15

4.3.1 Build-Time... 15

4.3.2 Run-Time 16

5 J2EE Connector Architektur Interfaces 18

5.1 Connection Management Interfaces 18

5.2 Common Client Interface(CCI) 23

5.3 Security Management Interfaces 28

5.4 Transaction Management Interfaces 30

6 Realisierung des Resource-Adapters für MQSeries Workflow 35

6.1 Aufbau einer Verbindung 35

6.2 Abbau einer Verbindung 40

6.3 Kommunizieren mit MQWF 42

6.4 Security Management 48

6.5 Transaction Management 48

7 Non-managed-Applikationsszenario 50

7.1 Die Struktur des Non-managed-Applikationsszenarios 50
7.2 Verwendung des Resource-Adapters 51
7.2.1 Erzeugung einer *ConnectionFactory*-Instanz 51
7.2.2 Registrieren einer *ConnectionFactory*-Instanz im JNDI-Namespace 51
7.2.3 Finden einer *ConnectionFactory*-Instanz 52
7.2.4 Registrierungsmechanismen 52
7.2.5 Erzeugung einer Verbindung zu einem EIS 55
7.2.6 Beenden einer Verbindung 56

8 Zeitmessungen des Resource-Adapters 58

8.1 Beispielszenario 58
8.2 Messungsumgebung 58
8.3 Messpunkte 59
8.4 Messergebnis 60
8.4.1 Der Gesamtzeitvergleich 60
8.4.2 Einzelne Zeitvergleiche 61
8.4.3 Abhängigkeit von Datenvolumina 65

9 Zusammenfassung und Ausblick 69

Abbildungensverzeichnis

Abbildung 2.1: Mehrschichtiges J2EE-Applikationsmodell ... 3
Abbildung 2.2: Client-Komponenten kommunizieren mit J2EE-Server ... 4
Abbildung 2.3: Business Tier und EIS Tier ... 5
Abbildung 2.4: Container Typen ... 6
Abbildung 3.1: Integration zwischen Application Server und EIS ... 8
Abbildung 3.2: Die Connector Architektur ... 9
Abbildung 4.1: Workflow System ... 14
Abbildung 4.2: Prozess-Diagram ... 15
Abbildung 4.3: Kommunikation zwischen Client und Server ... 16
Abbildung 5.1: Architektur-Diagramm über Connection-Management ... 18
Abbildung 5.2: Common Client Interface ... 24
Abbildung 6.1: Neue Verbindungserzeugung im Managed-Applikationsszenario ... 36
Abbildung 6.2: Wiederverwendung von Verbindungen im Managed-Applikationsszenario ... 37
Abbildung 6.3: Verbindungsbeenden im Managed-Applikationsszenario ... 41
Abbildung 6.4: Abbildung einer Datenstruktur auf einen Container ... 44
Abbildung 7.1: Die Architektur des Non-Managed-Applikationsszenarios ... 50
Abbildung 7.2: Verbindungserzeugung im Non-Managed-Applicationsszenario ... 56
Abbildung 7.3: Verbindungsbeenden im Non-Managed-Applikationsszenario ... 57
Abbildung 8.1: Die Messpunkte für die Messungen ... 60
Abbildung 8.2: Die Gesamtzeitmessungen für den Managed- und Non-Managed-Fall ... 61
Abbildung 8.3: Einzelne Zeitvergleiche für die erste Ausführung ... 62
Abbildung 8.4: Einzelne Zeitvergleiche für die stabilisierten Zustände ... 64
Abbildung 8.5: Die Zeitdauer für die Initialisierung in Abhängigkeit der Datenmenge ... 66
Abbildung 8.6: Die Zeit für die Abbildung in Abhängigkeit der Datenmenge ... 67
Abbildung 8.7: Die Zeit für die Ausführung in Abhängigkeit der Datenmenge ... 67

1 Einleitung

Viele komponenten-basierte Anwendungen werden entwickelt, um Geschäftsprozesse verschiedener Bereiche in der IT-Branche zu unterstützen. Solche Anwendungen greifen oft auf Enterprise-Information-Systeme(EIS) zu, um die benötigten Daten zu bekommen oder die Funktionen auf den EISs auszuführen. Wenn eine Anwendung von einem Applikationsserver auf verschiedene EIS zugreifen möchte, muss diese Anwendung zu jedem EIS angepasst werden. Es gab keine standardisierte Architektur, um die Integration zwischen einem Applikationsserver und einem EIS zu unterstützen.

Die J2EE-Connector-Architektur wird so definiert, dass das Problem gelöst werden kann. Um die Connector-Architektur zu unterstützen, muss ein EIS einen Resource-Adapter implementieren. Ein Applikationsserver muss erweitert werden, um die Connector-Architektur zu unterstützen. Der EIS-spezifische Resource-Adapter kann auf allen Applikationsservern installiert werden, die die Connector-Architektur implementiert, und bietet die zugrundliegende Infrastruktur, um die Integration zwischen einem Applikationsserver und dem EIS zu unterstützen. Das reduziert deutlich den Arbeitsaufwand, weil der Applikationsserver und das EIS nur einmal die Connector-Architektur implementieren müssen.

Aufgabenstellung

Die Aufgabe dieser Diplomarbeit ist es, einen prototypischen Resource-Adapter für das Workflow-Management-System MQSeries Workflow zu implementieren. Die Implementierung wird in Java realisiert. Zum Testen müssen auch eine J2EE-Anwendung und ein Workflow entwickelt werden. Am Ende soll das Laufzeitverhalten des Resource-Adapters in verschiedenen Fällen untersucht, und das Ergebnis analysiert werden.

Um die Aufgabe durchzuführen, muss man zuerst in die Themengebiete J2EE-Plattform, J2EE-Connector-Architektur und Workflow-Management-System einarbeiten. Schließlich soll ein konzeptioneller Entwurf für den Resource-Adapter erstellt wird. Dieser Resource-Adapter wird auf dem WebSphere-Applikationsserver installiert. Die J2EE-Anwendung soll über den Resource-Adapter einen Workflow auf dem MQSeries-Workflow-Server ausführen, und die zurückgelieferten Daten weiter verarbeiten.

Aufbau der Arbeit

Diese Arbeit teilt sich in 8 Kapiteln. In Kapitel 2 wird die J2EE-Architektur beschrieben. Auch die EJB-Technologie wird hier kurz erläutert, weil in dieser Diplomarbeit die EJB-Technologie verwendet wird.

Die Grundlagen der J2EE-Connector-Architektur wird im Kapitel 3 erklärt, auch die Motivation, warum ein EIS und ein Applikationsserver die Connector-Architektur implementieren soll, wird in diesem Kapitel begründet.

Im Kapitel 4 werden die Grundlagen eines Workflow-Management-Systems beschrieben. Die Grundkenntnisse über MQSeries Workflow werden auch in diesem Kapitel erklärt.

Im Kapitel 5 werden die Interfaces, die die Connector-Architektur definiert, genau im Detail erklärt. Wie ein Resource-Adapter in dieser Diplomarbeit implementiert wird, wird in Kapitel 6 erläutet.

Kapitel 7 zeigt, wie ein Resource-Adapter im Non-Managed-Applikationsszenario verwendet wird. Das heißt, ein Resource-Adapter wird nicht auf dem Applikationsserver installiert, sondern als eine Bibliothek von einem Java-Programm benutzt.

In Kapitel 8 wird beschrieben, wie die Zeitmessungen durchgeführt wurden, und auch das Messergebnis wird in diesem Kapitel analysiert. Kapitel 9 stellt eine Zusammenfassung und einen Ausblick dar.

2 Java 2 Platform, Enterprise Edition(J2EE)

Ein Ziel eines Unternehmens ist es, die hoch verfügbaren, zuverlässigen und skalierbaren Dienste für die Kunden anzubieten. Die meisten Enterprise-Dienste werden als mehrschichtige Applikationen implementiert. Diese Applikationen kombinieren normalerweise die existierenden Enterprise-Informations-Systeme(EIS) mit den neuen Business-Funktionen. Mittels Web-Technologien ermöglicht die erste Schicht der Applikation den Benutzern, auf die komplizierte Business-Funktionen zuzugreifen. Die mittlere Schicht integriert das EIS mit Business-Funktionen.

J2EE ermöglicht den Entwicklern, eine mehrschichtige Enterprise-Applikation zu entwickeln.

2.1 J2EE Architektur

Die J2EE-Plattform verwendet ein mehrschichtiges, verteiltes Applikationsmodell. J2EE Technologie basiert auf der Komponenten-Technologie. Eine J2EE Applikation besteht aus Komponenten, die auf verschiedenen Maschinen bzw. auf verschiedenen Schichten installiert werden.

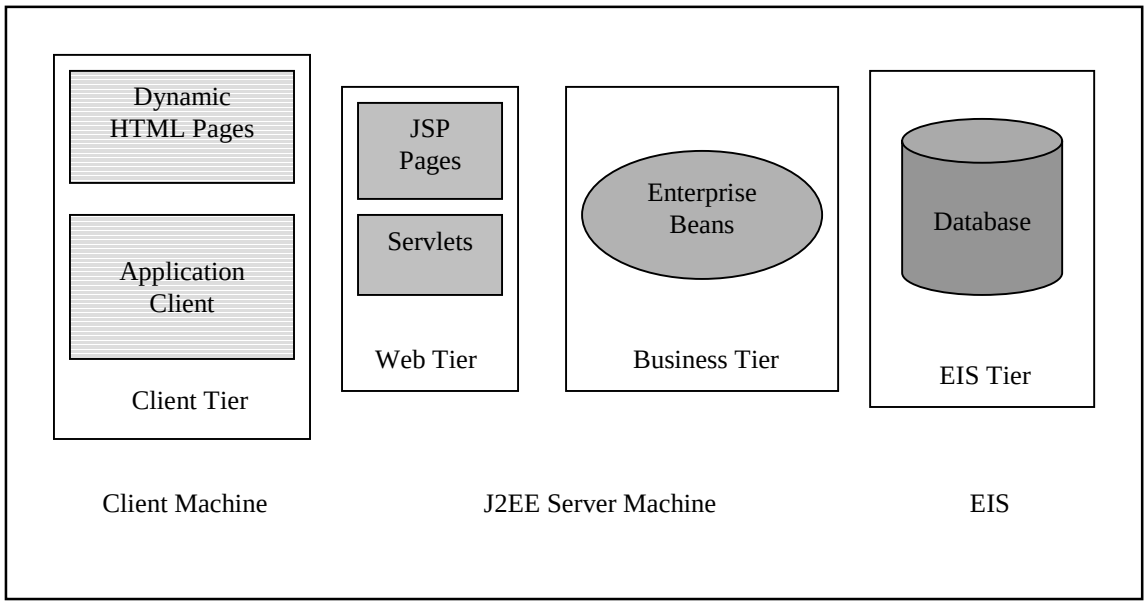


Abbildung 2.1[J2EEINTRO]: Mehrschichtiges J2EE-Applikationsmodell

Die Abbildung 2.1[J2EEINTRO] stellt das J2EE Applikationsmodell dar, das in mehrere Schichten aufgeteilt ist. Die erste Schicht ist der sogenannte Client-Tier, wobei Client-Tier-Komponenten auf der Client-Maschine laufen. Solche Client-Komponenten sind zum Beispiel Applikations-Clients oder Applets. Die zweite Schicht ist der Web-Tier. Web-Tier-Komponenten wie Java-Servlets oder Java-Server-Pages laufen auf den J2EE-Server. Zudem gibt es eine dritte Schicht, welche dann Business-Tier genannt wird. Enterprise JavaBeans sind Business-Komponenten, die auch auf J2EE-Servern laufen. Diese Komponenten sind alle

mit der Programmiersprache Java geschrieben, und stimmen mit der J2EE-Spezifikation überein. Der Enterprise-Information-System(EIS)-Tier ist die letzte Schicht. Die entsprechende Software läuft auf dem EIS-Server.

Client -Tier

Eine J2EE-Applikation kann entweder web-basiert oder nicht web-basiert sein. Für die nicht web-basierte J2EE-Applikationen wird ein Applikationsclient auf der Client-Maschine ausgeführt, während für eine web-basierte J2EE-Applikation ein Web-Browser die Web-Seite und Applets auf die Client-Maschine herunterlädt. Die Typen von Clients, die die J2EE Plattform unterstützt, sind in der Abbildung 2.2[J2EEINTRO] dargestellt.

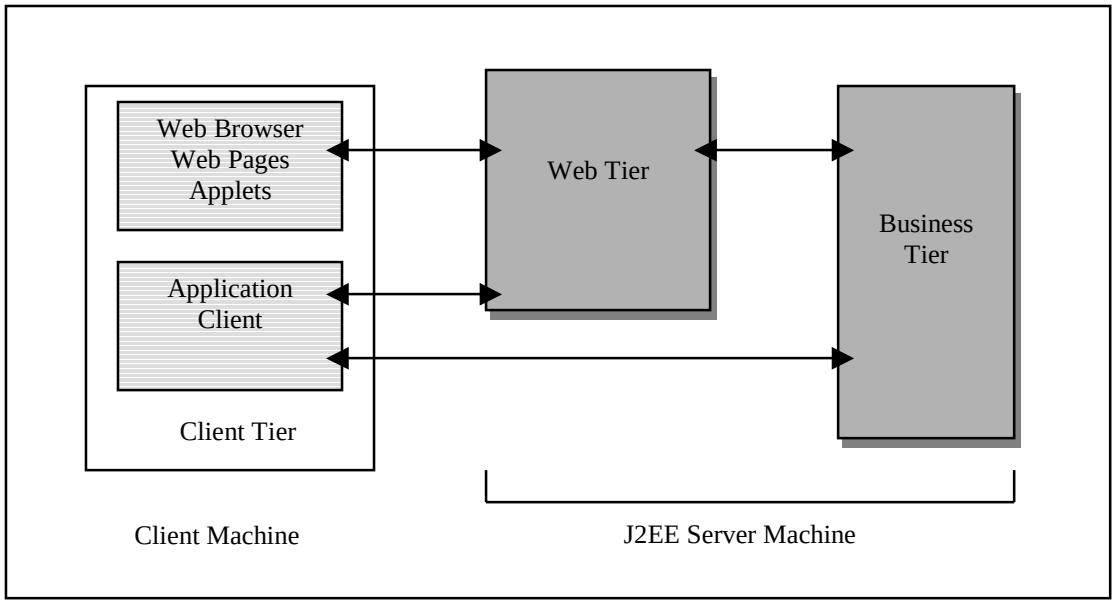


Abbildung 2.2[J2EEINTRO]: Client-Komponenten kommunizieren mit J2EE-Server

Applets sind die Graphical-User-Interface(GUI)-Komponenten. Sie werden vom Web-Tier heruntergeladen und typischerweise im Web-Browser ausgeführt. Applets können als ein Benutzer-Interface für J2EE-Applikationen verwendet werden. Sie können mit Java-Server-Pages(JSP)s und Servlets, die auf der Server-Seite laufen, kommunizieren und Business-Daten austauschen.

Ein Applikationsclient ermöglicht dem Benutzer auf einen J2EE-Server zuzugreifen. Applikationsclients können auf die Enterprise-Beans im Business-Tier direkt zugreifen. Aber der Applikationsclient kann auch eine HTTP-Verbindung zu einem Servlet im Web-Tier erzeugen.

Web-Tier

Der Web-Tier befindet sich auf dem J2EE-Server. Auf dem Web-Tier laufen Servlets und JSPs. Sie sind sogenannte J2EE-Web-Komponenten, und für HTTP-Requests vom Web-Client zuständig. Servlets und JSPs können eingesetzt werden, um eine HTML-Seite, die als Benutzer-Interface einer Applikation benutzt wird, dynamisch zu erzeugen, oder um XML zu erzeugen, die von anderen Applikationskomponenten verwendet werden. Servlets sind reine

Javaprogramme, während JSPs sowohl HTML-Code als auch Java-Code enthalten. Ein JSP wird zu einem Servlet kompiliert.

Business-Tier

Auf dem J2EE-Server befindet sich noch das Business-Tier. Enterprise-Beans laufen auf dem Business-Tier. Ein Enterprise-Bean enthält typischerweise Business-Logik für eine J2EE-Applikation. Ein Enterprise-Bean bekommt die Daten von dem Client –Programm, bearbeitet sie und schickt sie dem Enterprise-Information-System, um sie dort zu speichern. Oder es holt sich die Daten vom EIS und schickt sie dem Client-Program nach der Bearbeitung. Dieser Vorgang ist in der Abbildung 2.3[J2EEINTRO] dargestellt.

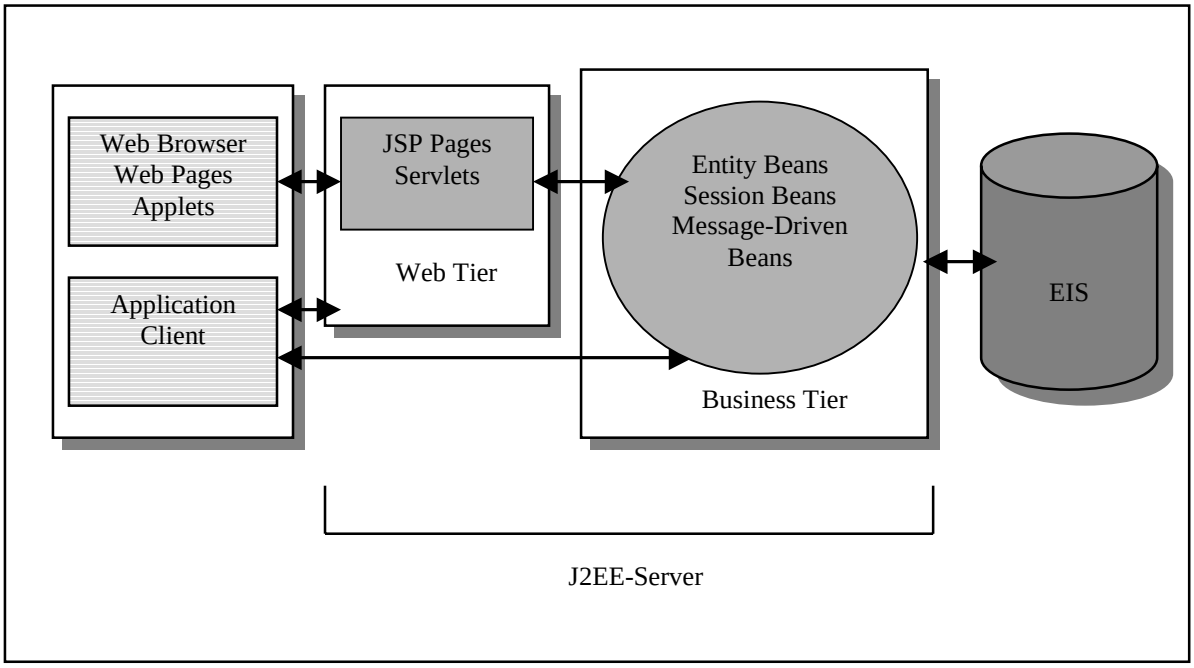


Abbildung 2.3[J2EEINTRO]: Business Tier und EIS Tier

EIS Tier

Das Enterprise-Information-System umfasst Enterprise-Infrastructure-Systeme wie Enterprise-Resource-Planning(ERP), Database-Systeme, Mainframe-Transaction-Processing usw. Zum Beispiel, J2EE-Applikationskomponenten greifen auf das Enterprise-Information-System zu, um mit einer Datenbank eine Verbindung zu erstellen.

Container

Ein Container bietet eine Laufzeit-Umgebung für die J2EE-Applikationskomponenten. Die Komponenten benutzen die Protokolle und die Schnittstellen des Containers, um mit den Plattform-Diensten oder miteinander zu interagieren. Die Dienstanforderungen werden im Deployment-Descriptor beschrieben. Zur Laufzeit wird der Container diese Anforderung einlesen und erfüllen. Solche Basis-Dienste sind zum Beispiel Transaktionsmanagement,

Sicherheitsmanagement, Java Naming and Directory Interface(JNDI) und Remote-Verbindungsaufbau.

Ein typisches J2EE-Produkt bietet einen Container für jeden Typ von Applikationskomponente: Applikationsclient-Container, Applet-Container, Web-Komponenten-Container und Enterprise-JavaBeans-Container, so wie sie in der Abbildung 2.4[J2EEINTRO] dargestellt sind.

Web-Container und Enterprise-JavaBeans-Container befinden sich auf J2EE-Server-Seite. Ein Enterprise-JavaBeans-Container verwaltet die Ausführung der Enterprise-Java-Beans. Ein Web-Container ist für die Ausführung der JSP-Pages und Servlets zuständig.

Auf Client-Seite gibt es Applet-Container und Applikationsclient-Container. Ein Applet-Container ist die Kombination aus Web-Browser und Java Plug-In. Ein Applikationsclient-Container verwaltet die Ausführung der Applikationsclient-Komponenten.

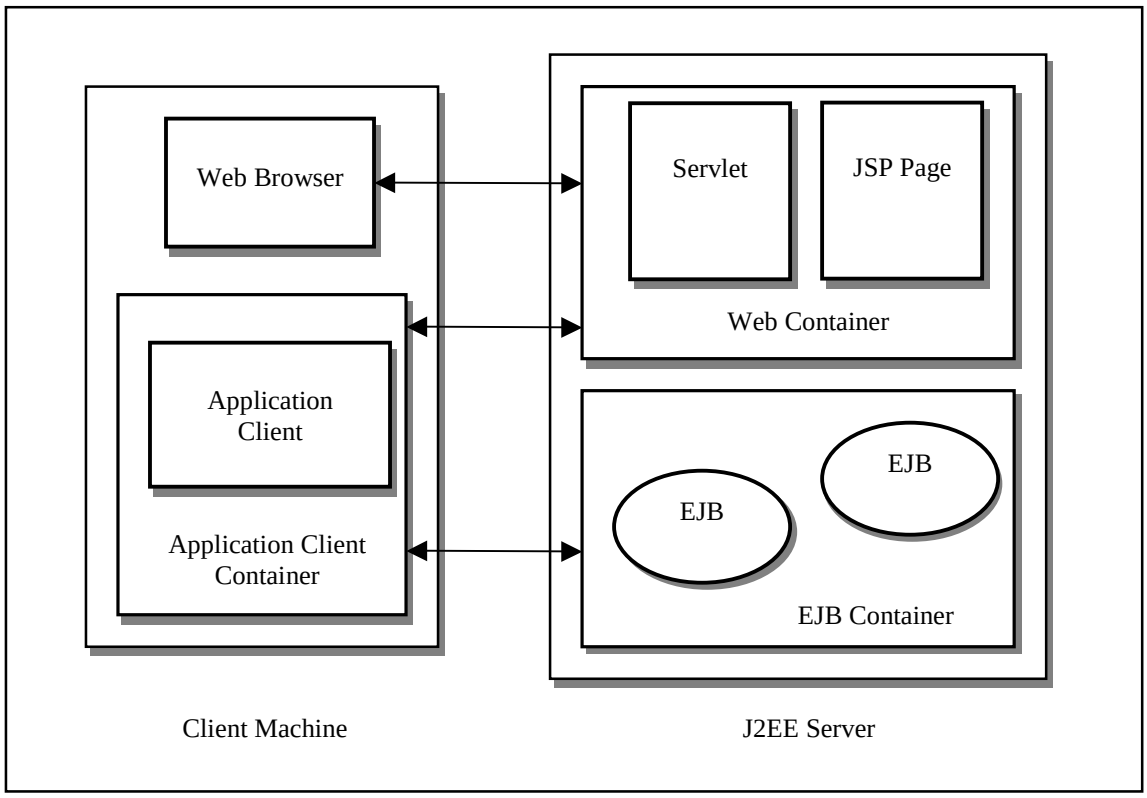


Abbildung 2.4[J2EEINTRO]: Container Typen

2.2 Enterprise JavaBeans

Enterprise Beans sind verteilte serverseitige Komponenten. Sie passen sich einem Standard-Komponenten-Modell an und können ohne Modifikation auf einem Server direkt ausgeführt werden.

Eine EJB-Komponente beinhaltet die Enterprise-Bean-Klasse, das Remote-Interface, das Home-Interface, den Primary Key (nur für Entity Beans), den Deployment-Descriptor und das

EJB-Jar-File. Die Enterprise-Bean-Klasse implementiert die Geschäftslogik des Beans. Das Home-Interface definiert die Methoden für die Erzeugung, das Laden und die Zerstörung eines Beans. Das Remote-Interface definiert die Geschäftsmethode des Beans. Ein EJB-Objekt implementiert das Remote-Interface und ein EJB-Home-Objekt implementiert das Home-Interface. Der Client fordert ein Bean von EJB-Home-Objekt an. Dann wird eine Instanz eines EJB-Objekts von dem EJB-Home-Objekt erzeugt, das das entsprechende Bean referenziert. Danach kann der Client die Methoden eines Beans aufrufen. Der Primary Key ist eine Java-Klasse, die einen Zeiger zur Datenbank bietet. Nur Entity Beans brauchen einen Primary Key, denn sie repräsentieren die Daten in der Datenbank. Ein Bean-Anbieter deklariert Diensteanforderungen für seine Komponenten in einer Deployment-Descriptor-Datei, um dem Container seine Dienste-Anforderungen mitzuteilen. Der Container untersucht den Deployment-Descriptor und erfüllt die Anforderungen.

Es gibt drei Typen von Enterprise JavaBeans: Session Beans, Entitiy Beans und Message-Driven Beans. Session Beans sind Geschäftsprozess-Objekte. Sie implementieren eine Geschäftslogik. Es gibt zwei Subtypen von Session Bean — stateful session beans und stateless session beans. Entity Beans repräsentieren persistente Daten. Message-Driven Bean kombinieren die Eigenschaft des Session-Beans und die Eigenschaft von Message-Listenern der JMS-Messages. Sie erlauben einer Business-Komponente, JMS-Messages asynchron zu empfangen und zu behandeln. Hier in dieser Diplomarbeit wird ein Session-Bean zum Testen implementiert.

3 J2EE Connector-Architektur

3.1 Overview

Die J2EE-Connector-Architektur(Final release 1.0) definiert eine standardisierte Architektur, um die J2EE-Plattform mit dem heterogenen EIS zu verbinden, zum Beispiel ERP, Mainframe-Transaction-Processing(TP) und Datenbank-Systeme.

Es gab keine standardisierte Architektur für die Integration zwischen einem Applikationsserver und EISs. Viele EIS- und Applikationsserver-Vendor benutzen eine Vendor-spezifische Architektur, um EIS-Integration zu unterstützen. Das heißt, ein EIS-Vendor muss sich an jeden Applikationsserver anpassen, und ein Applikationsserver muss den angepassten Code hinzufügen , wenn er ein neues EIS unterstützen möchte.

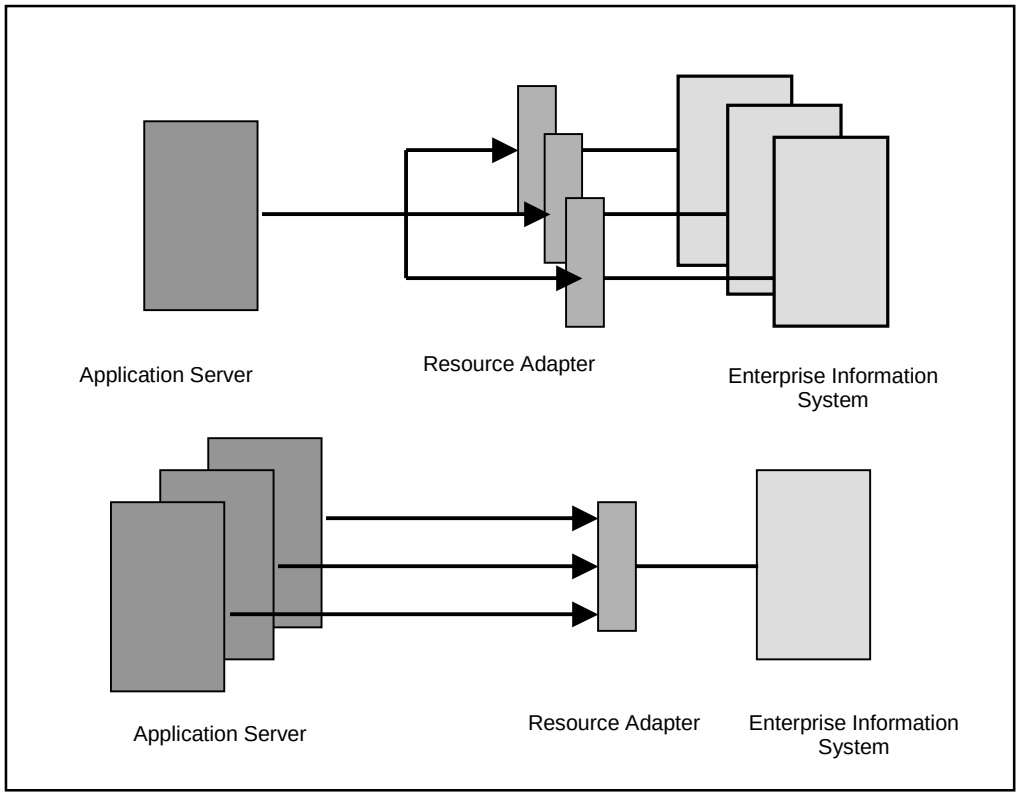


Abbildung 3.1[J2EERA]: Integration zwischen Application Server und EIS

Die Connector-Architektur ermöglicht einen EIS-Vendor, ein Resource-Adapter für seines EIS anzubieten; Das Resource-Adapter wird im Applikationsserver installiert und bietet die zugrundliegende Infrastruktur für die Integration zwischen dem EIS und dem Applikationsserver. Ein Applikationsserver-Vendor muss nur einmal den Applikationsserver erweitern, um die Connector-Architektur zu unterstützen. Der EIS-Vendor muss nur einmal das standardisierte Resource-Adapter implementieren, das mit jedem Applikationsserver, der die standardisierte Connector-Architektur unterstützt, angeschlossen werden kann. Abbildung 3.1[J2EERA] stellt die Integration zwischen den Applikationsservern und den EISs dar.

Die Connector-Architektur definiert auch ein Common-Client-Interface(CCI) für den Zugriff auf das EIS. Das CCI definiert eine Client-API, um mit heterogenem EIS zu kommunizieren.

3.2 JDBC API und Common-Client-Interface

Das JDBC-API und CCI bieten beide die Schnittstelle für die Applikationskomponente, damit sie auf das zugrundliegende EIS zugreifen können.

Das JDBC-API definiert ein standardisiertes Java-API für den Zugriff auf relationale Datenbanken. Dieses Java-API schickt der Datenbank ein SQL-Statement und bearbeitet die Daten, die von der Datenbank zurückgeliefert werden. Das JDBC-API ist empfohlen für den Zugriff auf relationale Datenbanken.

Das Common-Client-Interface ist definiert von der standardisierten J2EE-Connector-Architektur, sie bietet den Applikationskomponenten die Möglichkeit, auf verschiedene EISs zuzugreifen. Sie sind empfohlen für den Zugriff auf EISs.

3.3 Connector-Architektur

Wie in der Abbildung 3.2[CONSPE10] illustriert wird, ist die Connector-Architektur in einem Applikationsserver und in einem EIS-spezifischen Resource-Adapter implementiert. Ein Resource-Adapter ist eine zentrale Komponente für die Integration und der Verbindung zwischen einem Applikationsserver und einem EIS. Es ist ein Software-Treiber, der von den Applikationen verwendet wird, um eine Verbindung zu einem EIS zu erzeugen und zu beenden. Der Resource-Adapter wird im Applikationsserver installiert und bietet die Verbindung zwischen dem EIS, dem Applikationsserver und den Applikationskomponenten.

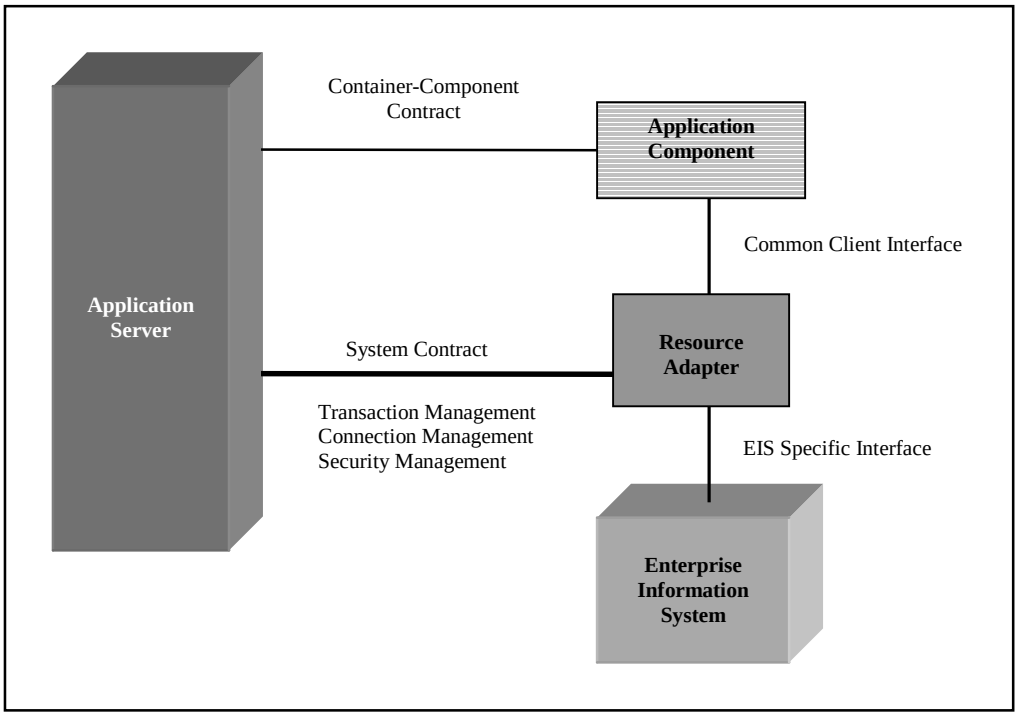


Abbildung 3.2[CONSPE10]: Die Connector Architektur

Ein Applikationsserver und ein Resource-Adapter zusammen führen alle Mechanismen auf der Systemebene durch, zum Beispiel Transaktions-Management, Sicherheit-Management und Connection-Management, transparent zu den Applikationskomponenten. Dies hat den Vorteil, dass die Applikationskomponenten-Entwickler sich auf die Business-Logik und die Präsentations-Logik konzentrieren können und sich nicht um die Dienste auf Systemebene und EIS-spezifische Schnittstellen und Paradigmen kümmern müssen. Das reduziert deutlich die Zeit und den Aufwand.

Die J2EE Connector-Architektur definiert die Verträge, mit denen der Resource-Adapter mit den anderen Applikationskomponenten und dem Applikationsserver kommunizieren. Zwischen dem Resource-Adapter und den Applikationsserver existiert der System-Contract, dazu gehören Transaction-Management, Security-Management und Connection-Management. Zwischen den Applikationskomponenten und dem Resource-Adapter ist das Common-Client-Interface definiert. Das Common-Client-Interface bietet ein Client-API, mit dem die Java-Applikationen auf den Resource-Adapter zugreifen können.

3.3.1 System-Contracts

Connection-Management-Contract

Das Connection-Management unterstützt Connection-Pooling, um die Verbindungen zu verwalten und die Kosten für die Verbindungserzeugung zu sparen. Wenn eine Applikationskomponente eine Verbindung zu einem EIS erstellen möchte, wird der Applikationsserver zuerst im Pool eine passende Verbindung suchen, wenn es eine passende Verbindung gibt, wird diese Verbindung benutzt. Wenn es keine passende Verbindung gibt, wird eine neue Verbindung erzeugt und in das Pool hinzugefügt. Das optimiert die Durchfühung der Applikationen und erhört die Skalierbarkeit. Der Connection-Pooling-Mechanismus soll transparent für die Applikationen sein. Das vereinfacht die Entwicklung der Applikationen.

Ein Applikationsserver benutzt das Connection-Management, um eine Verbindung zu einem EIS zu erstellen, die Connection-Factory im JNDI-Namespace zu konfigurieren, und eine passende Verbindung in einem Connection-Pool zu finden.

Transaction-Management-Contract:

Es gibt zwei Arten von Transaktionen, nämlich lokale Transaktion und globale Transaktion(XA Transaktion). Eine lokale Transaktion beschränkt sich auf ein einzelnes EIS. Der Resource-Manager des EISs verwaltet selbst die Transaktion. Eine XA Transaktion kann mehrere Resource-Manager des EISs umfassen. Dazu ist ein externer Transaktionsmanager notwendig. Dieser Transaktionsmanager ist typischerweise mit einem Applikationsserver zusammengebündelt und benutzt ein Two-Phase-Commit Protokoll.

Die Connector-Architektur definiert einen Transaction-Management-Contract zwischen einem Applikationsserver und einem Resource-Adapter. Der Transaction-Management-Contract unterstützt die lokale Transaktionen und die XA Transaktionen. Dieser Kontrakt ermöglicht den Applikationsserver, die Infrastruktur und Runtime-Umgebung für das Transaktionsmanagement anzubieten. Die Applikationskomponenten verlassen sich auf diese

Transaktionsinfrastruktur. Ein Resource-Adapter kann die folgenden Transaktionseigenschaften bieten:

- Keine Unterstützung für Transaktionen
- Unterstützt nur lokale Transaktionen
- Unterstützt sowohl lokale Transaktionen als auch XA Transaktionen

Ein Applikationsserver muss Transaktionen auf allen Ebenen unterstützen. Das gewährleistet, dass der Applikationsserver den EISs auf verschiedene Transaktionsebenen unterstützen kann.

Security-Management-Contract

Für ein Unternehmen ist es sehr wichtig, dass die Informationen in einem EIS geschützt werden, damit nur autorisierter Zugriff auf EIS erlaubt ist. Um eine Verbindung zu einem EIS zu erstellen, muss ein EIS-Sign-On durchgeführt werden. Dafür sollen die folgenden Schritte durchgeführt werden:

- Benutzer Identifikation und Authentifizierung
- Autorisieren und Access-Control
- Kommunikationssicherheit zwischen dem Applikationsserver und dem EIS gewährleisten

Für die Authentifizierung unterstützt die Connector-Architektur zwei Mechanismen, nämlich *BasicPassword* und *Kerbv5*(Kerberos version 5 basierte Authentifizierungsmechanismus).

Die Autorisierung kann entweder auf dem Applikationsserver oder auf dem EIS überprüft werden, oder sogar auf beiden. Die Autorisierung auf einem EIS kann auf eine EIS-spezifische Weise geprüft werden, zum Beispiel, kann ein EIS eine Access-Control-Liste definieren, in der die Rollen und die zugehörige Erlaubnis definiert werden. Die Autorisierung kann auch auf Applikationsserver überprüft werden, zum Beispiel, erlaubt ein Applikationsserver einem Benutzer, eine Verbindung zu einem EIS zu erzeugen, nur wenn der Benutzer entsprechende Rechte besitzt.

Die Kommunikationssicherheit kann durch verschiedene Protokolle gewährleistet werden. Um die Kommunikationen auf einem unsicheren Kommunikationskanal zu schützen, können die Protokolle zum Beispiel Kerberos verwenden, die Authentifizierung, Integrität und Vertrauenswürdigkeit unterstützt. Kommunikationen können auch durch einen sicheren Kommunikationskanal zum Beispiel SSL geschützt werden.

3.3.2 Common-Client-Interface(CCI)

Das CCI ist ein standardisiertes Interface für die Applikationskomponenten. Durch CCI kann eine Applikation im JNDI-Namespace eine Connection-Factory finden. Von einer Connection-Factory kann sie eine Verbindung zu einem EIS erzeugen. Eine Applikation benutzt die Interaktion, die das CCI anbietet, um die Daten an das EIS zu übergeben und auf das EIS zuzugreifen. Das CCI bietet den Applikationen auch die Möglichkeit, die Verbindung zu dem EIS abzubauen.

3.3.3 Non-Managed-Applikationsszenario

Die Connector-Architektur unterstützt nicht nur den Zugriff einer Applikationskomponente, sondern auch den Zugriff eines Non-Managed-Applikationsclients (z.B. Java-Applikationen und Java-Applets) auf ein EIS.

In diesem Fall wird der Resource-Adapter nicht auf dem Applikationsserver installiert. Der Applikationsclient verwendet den Resource-Adapter als eine Bibliothek. Er ist selbst für die Sicherheit und die Transaktion zuständig.

4 Workflow Management System(WFMS)

Ein Workflow-Management-System bietet die Möglichkeit, immer wiederkehrende Abläufe innerhalb eines Unternehmens effektiv und automatisch zu steuern, zum Beispiel den Kreditaufnahmeprozess in einer Bank, der Genehmigung von Urlaubsanträgen, usw. Da solche Prozesse immer wieder stattfinden, bietet ein einmal definierter Workflow enorme Zeitersparnis.

4.1 Grundbegriffe

Ein Geschäftsprozess ist eine Reihe von Aktivitäten, die von verschiedenen Personen ausgeführt werden. Wie zum Beispiel das Versicherungsprozess in einer Versicherungsfirma. Solche Aktivitäten werden immer wiederholt.

Eine Prozess-Definition ist die Repräsentation eines Geschäftsprozesses, sie beschreibt die Struktur eines Geschäftsprozesses[WRM95]. Eine Prozess-Definition besteht aus den Aktivitäten, der Reihenfolge der Aktivitäten, den Bedingungen, ein Prozess zu starten und zu beenden, und den Informationen über die Aktivitäten, z.B. IT-Applikation, Teilnehmern usw.. Es ist möglich, dass eine Prozess-Definition aus manuellen und automatischen(workflow) Aktivitäten besteht.

Ein Workflow ist der Teil von der Prozess-Definition, der von dem Computer unterstützt wird[WRM95]. Er beschreibt eine Folge von Aktivitäten, wobei die Folge durch Ereignisse ausgelöst und beendet wird. Ein Workflow kann, abhängig von Bedingungen, ganz oder in Teilen alternativ ausgeführt werden. Er kann parallel und sequentiell ausgeführt werden.

Ein Workflow-Management-System definiert die Prozess-Definition, erzeugt die Instanz des Prozess-Definitions, verwaltet die Prozess-Instanzen und ihre Ausführung, kontrolliert das Interagieren mit den Teilnehmern und den Applikationen.

Workflows haben drei Dimensionen: *Process-logic*, *organisation* und *IT-infrastructure* [FLDR].

Process-logic beschreibt die Aktivitäten und die Reihenfolge von den Aktivitäten, in der die Aktivitäten ausgeführt werden. Die Aktivitäten können sequentiell oder parallel ausgeführt werden.

Organisation beschreibt die Struktur der Organisation in einer Firma. Sie enthält die Informationen über die Abteilungen, die Rollen und die Personen. Diese Informationen werden benutzt, um zu definieren, wer welche Aktivität ausführt.

IT-Infrastruktur beschreibt, welche IT-Ressourcen benötigt werden, wie zum Beispiel, die Programme, die die Aktivitäten ausführen.

4.2 Standardisierte Workflow-Management-System

Es gibt viele Firmen, die Workflow-Management-Systeme anbieten. Um die verschiedene WFMS von verschiedenen Firmen zusammenarbeiten zu können, definiert Workflow-Management(WFM)-Coalition die standardisierte Struktur eines WFMSs. Diese Struktur ist in der Abbildung 4.1[WRM95] dargestellt.

Der erster Teil ist die *Build-Time*-Komponente. In der *Build-Time*-Komponente kann die *Process-Definition* definiert werden, welche auch als *process-template*, *process-metadata* oder *process-model* bezeichnet wird[WRM95].

In einer *process-definition* werden sowohl die Aktivitäten, die entweder mit den Programmen oder mit den Personen verbunden sind, definiert, als auch die Reihenfolge, in der die Aktivitäten ausgeführt werden, definiert. Die *process-definition* kann im Text, in grafischer Form oder in einer *formal-language-notation* beschrieben werden[WRM95].

Der zweite Teil sind die *Run-Time-control-functions*[WRM95]. Die Kern-Komponente ist die *workflow-management-control-software*(oder *Engine*). Sie ist zuständig für die Erzeugung und für die Löschung eines Prozesses, die Kontrolle von der Aktivitätsplanung während eines Prozesses, und das Interagieren mit Applikationstools oder menschlichen Ressourcen.

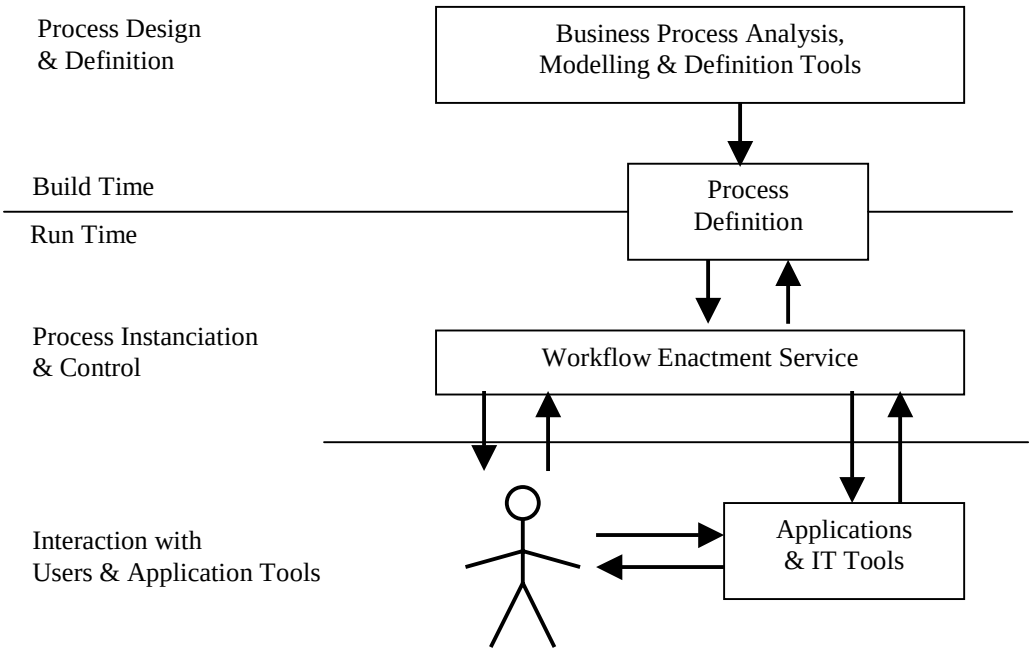


Abbildung 4.1[WRM95]: Workflow System

Der dritte Teil sind die *Run-Time-Activity-Interactions*[WRM95]. Jede Aktivität in einem Workflow-Prozess ist typischerweise mit menschlichen Operationen oder Applikationsprogrammen verbunden. Das Interagieren mit der *process-control-software* ist notwendig, um die Kontrolle zwischen den Aktivitäten zu übertragen, die Prozesszustände zu ermitteln, die Applikationstools aufzurufen und die entsprechende Daten zu übertragen.

4.3 MQ Series Workflow(MQWF)

IBM-MQWF besteht aus zwei Komponenten: der Build-Time-Komponente und der Run-Time-Komponente. Die Build-Time-Komponente in MQWF entspricht der Build-Time-Komponente des standardisierten Workflow-Management-Systems, welche in Abbildung 4.1 veranschaulicht wird. Die Run-Time-Komponente in MQWF entspricht genau zwei Teilen des standardisierten Workflow-Management-Systems, nämlich den *Run-Time-control-functions* und den *Run-Time-Activity-Interactions*.

4.3.1 Build-Time

Mit der Build-Time-Komponente können die Prozess-Templates und die System-Ressourcen definiert werden. Die Build-Time-Komponente bietet einen graphischen Editor, in dem die Prozess-Templates erstellt werden können. In der Build-Time-Komponente können auch die Organisation, die Programme und das Netzwerk definiert werden. Die Prozess-Templates im graphischen Editor können in der MQ Workflow-Definition-Language(FDL)-Datei exportiert werden.

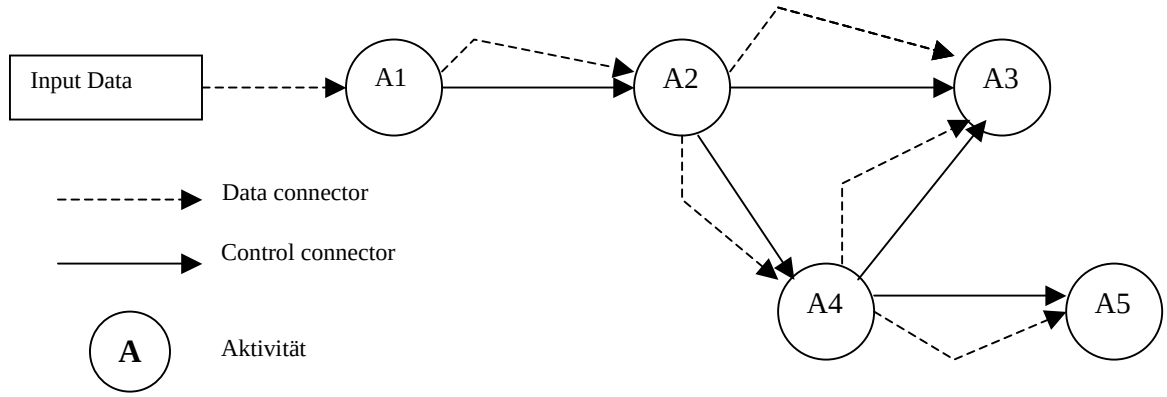


Abbildung 4.2: Prozess-Diagram

Abbildung 4.2 stellt ein Beispiel für ein Prozess-Diagramm dar. Das Diagramm besteht aus den Aktivitäten, *Data-Connector* und *Control-Connector*. Eine Aktivität kann mit einem Programm verbunden werden. Das Programm muss im Build-Time registriert werden, damit die Aktivitäten im Run-Time auf das Programm zugreifen können. Das Programm soll lauffähig sein. Es kann mit der Programmiersprache Java, C oder C++ usw. geschrieben werden. *Data-Connector* spezifiziert den Datenfluss von einer Aktivität zu anderer Aktivität. *Control-Connector* spezifiziert die Reihenfolge von den Aktivitäten in einem Prozess.

Prozesse und Aktivitäten haben entsprechende Input- und Output-Container. ein Input-Container einer Aktivität ist nur für diese Aktivität verwendbar. Wenn eine Aktivität die Daten von dem Output-Container der vorhergehenden Aktivität, oder von dem Input-Container des Prozesses brauchen, müssen die Daten von dort kopiert werden. Wenn die Datenstruktur von einem Output-Container einer Aktivität anders als die Datenstruktur von dem Input-Container einer anderen Aktivität ist, dann ist ein Container-Map notwendig. Das Map spezifiziert, welches Feld im Output-Container einer Aktivität welchem Feld im Input-Container einer anderen Aktivität entspricht. Jeder Container ist mit einer Daten-Struktur

verbunden. Eine Daten-Struktur kann auf einer vorher definierten Daten-Struktur, oder auf einer der folgenden Basis-Daten-Typen basieren: String, Binary, Float, Long usw.

Im Build-Time können auch die Personen definiert werden, die für die Ausführung des Prozesses oder der Aktivitäten zuständig sind. Man kann auch für die Aktivitäten Rollen definieren, und Personen zu den Rollen zuweisen. Eine Person kann mehrere Rollen übernehmen. Eine Rolle kann auch mehreren Personen zugewiesen werden. Das ist flexibel, weil eine Aktivität nicht fest mit einem Person verbunden ist.

Nach der Erstellung soll ein Prozess-Template von Build-Time in eine Text-Datei exportiert werden. Diese Text-Datei ist im FDL-Format. Die Datei soll dann in die Runtime-Datenbank importiert werden, damit ein Prozess-Template im Run-Time erzeugt wird. Das Prozess-Template wird von einem MQWF-Client gestartet und von den Server-Komponenten verwaltet.

4.3.2 Run-Time

Im Buildtime wird ein Prozess-Template definiert. Im Runtime kann die Instanzen von dem Prozess-Template erzeugt und ausgeführt werden. MQ Workflow bietet verschiedene API, zum Beispiel C-API, C++-API und Java-API. Mit dem API kann ein Workflow-Client implementiert werden, um mit den Server-Komponenten zu kommunizieren, und die Prozess-Instanzen des Prozess-Templates zu erzeugen, auszuführen und dessen Ausführungen zu überwachen. Hier in dieser Diplomarbeit wird nur das Java-API verwendet.

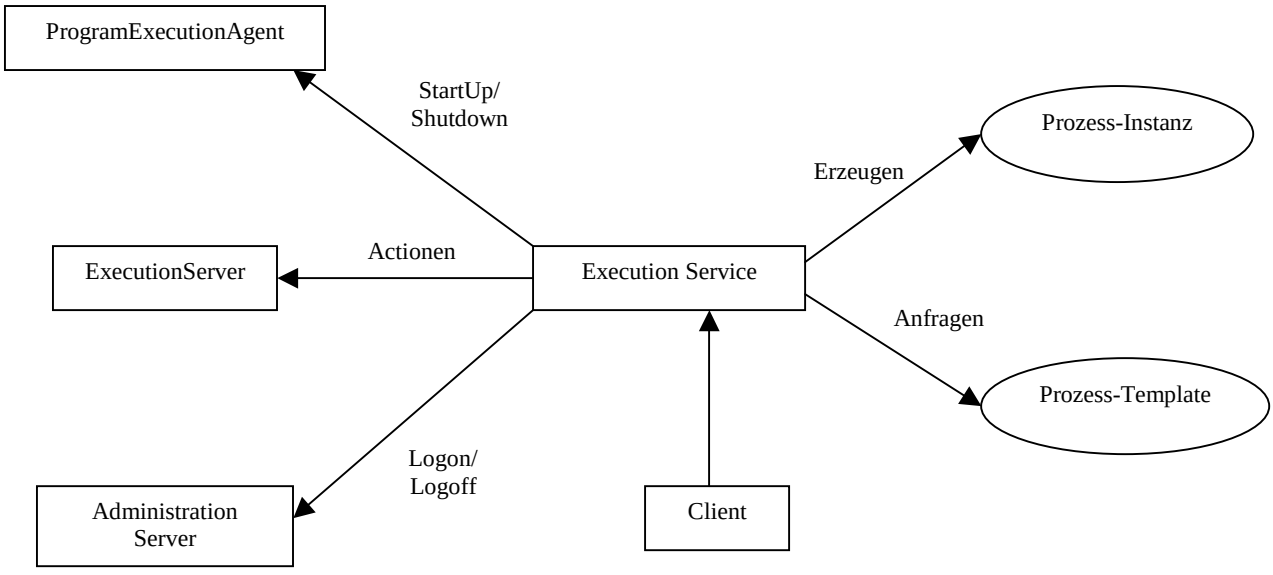


Abbildung 4.3[IBMWFPG]: Kommunikation zwischen Client und Server

Um die Runtime-Services zu nutzen, muss eine Kommunikation zwischen dem Client und einem MQWF Execution-Server aufgebaut werden. Der Execution-Server ist für die Ausführung der Prozess-Instanzen zuständig. Er interpretiert die *process-definitions*, erzeugt die Prozess-Instanz und startet dann, stoppt oder unterbricht sie. Er navigiert zwischen den

Aktivitäten, ruft die Programme oder die Personen auf, die für diese Aktivität zuständig sind. Um mit dem Execution-Server zu kommunizieren, muss zuerst ein ExecutionService erhalten werden, wie es in der Abbildung 4.3[IBMWFPG] gezeigt wird.

Ein ExecutionService-Object repräsentiert eine Session zwischen einem Benutzer und einem MQWF Execution-Server. Um die Session aufzubauen, muss sich der Benutzer auf den Administrationsserver einloggen, und von dem Server authentifiziert werden.

Wenn die Session zu einem Execution-Server schon aufgebaut wurde, kann der Client verschiedene Actionen auf dem Execution-Server durchführen. Zum Beispiel kann der Client auf die Prozess-Templates, und die Prozess-Instanzen zugreifen, die dem eingeloggten Benutzer autorisieren. Der Client kann auch den Program-Execution-Agent(PEA) starten, um die Aktivitäts-Programme auszuführen. Wenn ein Aktivitäts-Programm so konfiguriert wurde, dass es automatisch ausgeführt werden soll. Dann werden die Inputdaten automatisch zu dem PEA geschickt, und der PEA startet dieses Programm. Das Aktivitäts-Programm kann auf den PEA auf die Input- und Output-Container zugreifen.

5 J2EE Connector Architektur Interfaces

In der J2EE Connector Architektur werden die Interfaces und Klassen für das Common-Client-Interface und den System-Contract definiert. Der System-Contract enthält den Connection Management Contract, Transaction Management Contract und Security Contract.

5.1 Connection Management Interfaces

Der Connection-Management-Contract spezifiziert den Vertrag zwischen einem Applikationsserver und einem Resource-Adapter.

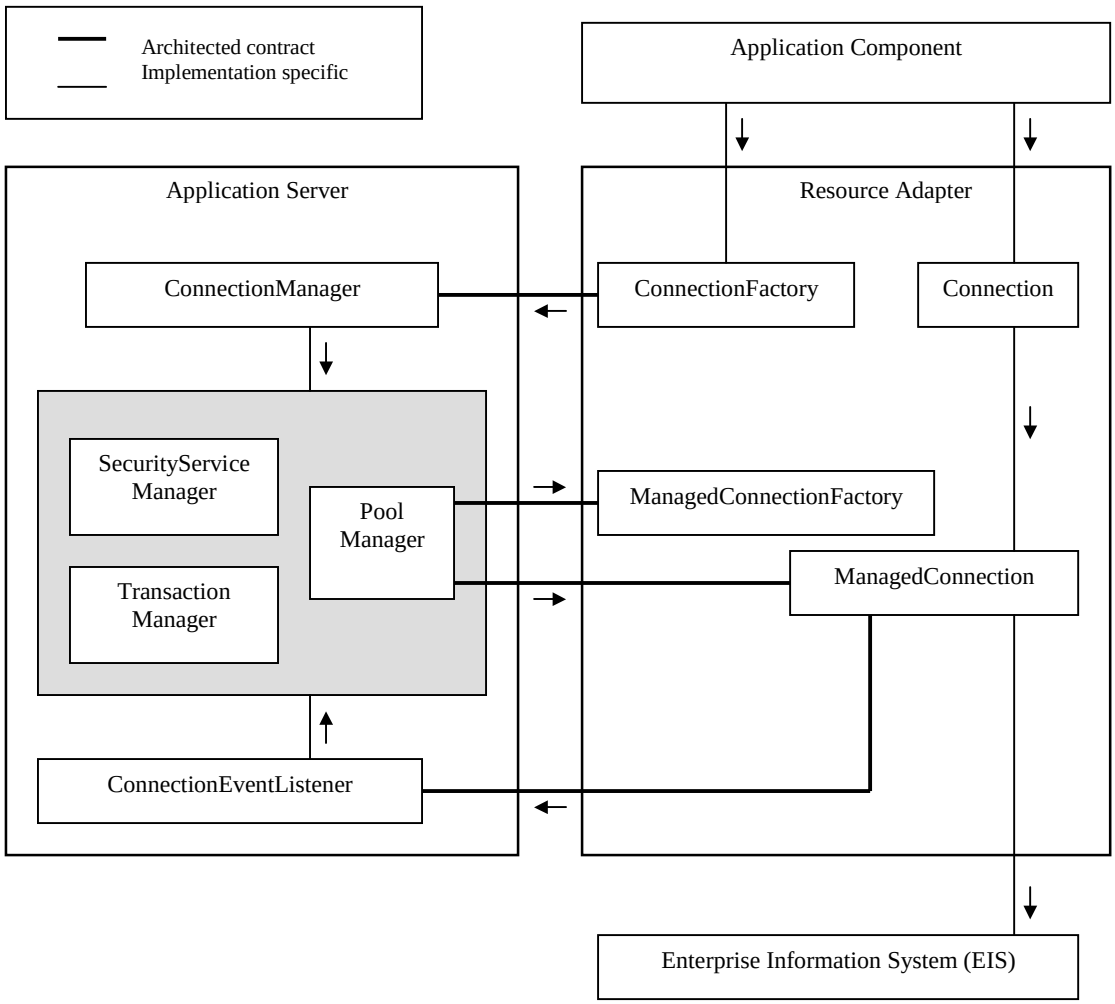


Abbildung 5.1[CONSPE10]: Architektur-Diagramm über Connection-Management

Die Abbildung5.1[CONSPE10] zeigt, wie eine Applikationskomponente mit einem EIS interagiert. Der Resource-Adapter bietet der Applikationskomponente die Interfaces *javax.resource.cci.Connection* und *javax.resource.cci.ConnectionFactory*. Durch die beiden

Interfaces kann eine Applikationskomponente eine Verbindung zu einem EIS erzeugen, und mit dem EIS kommunizieren.

Eine Applikationskomponente sucht eine *ConnectionFactory*-Instanz im JNDI-Namespace. Dann benutzt sie diese Instanz, um eine Verbindung zu einem EIS zu erstellen. Die *ConnectionFactory*-Instanz schickt diese Verbindungserzeugungsanfrage an eine *ConnectionManager*-Instanz. Das Interface *ConnectionManager* kann von einem Applikationsserver oder einem Resource-Adapter implementiert werden. Wenn ein Applikationsserver das Interface implementiert, bietet er auch die Dienste Transaction-Management, Sicherheits-, Error-Logging und -Tracing, und Connection-Pool-Management für die Applikationenkomponenten an. Wenn eine *ConnectionManager*-Instanz eine Verbindungserzeugungsanfrage von einer *ConnectionFactory*-Instanz bekommt, sucht sie eine passende Verbindung in dem Connection-Pool. Falls es keine passende Verbindung gibt, benutzt der Applikationsserver das Interface *ManagedConnectionFactory*, um eine physikalische Verbindung zu dem EIS zu erzeugen. Das heißt, eine *ManagedConnection*-Instanz wird erzeugt und zurückgeliefert. Diese neue *ManagedConnection*-Instanz wird in den Connection-Pool hinzugefügt. Falls der Applikationsserver eine passende Verbindung findet, dann liefert er eine *ManagedConnection*-Instanz zurück. Der Applikationsserver registriert eine *ConnectionEventListener*-Instanz an einer *ManagedConnection*-Instanz, um die Meldungen(z.B. Connection-Close, Transaction-Start usw.) zu bekommen, die mit einer *ManagedConnection*-Instanz verbunden sind. Der Server benutzt diese Meldungen, um zum Beispiel den Connection-Pool zu verwalten, worauf später noch eingegangen wird.

ManagedConnectionFactory

Das *javax.resource.spi.ManagedConnectionFactory*-Interface mit folgenden Methoden muss von einem Resource-Adapter implementiert werden.

```
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();
}
```

Beim Deployment eines Resource-Adapters wird die Methode *createConnectionFactory()* vom Depolymentstool aufgerufen, um eine *ConnectionFactory*-Instanz zu erzeugen, die mit einem JNDI-Name verbunden ist, zum Beispiel eis/WFAdapter.

Das *ManagedConnectionFactory*-Interface unterstützt Connection-Pooling. Es definiert die Methoden für die Verbindungserzeugung und die Verbindungsanpassung. Wenn der Applikationsserver eine Verbindung zu einem EIS erzeugen möchte, fragt er eine *ManagedConnectionFactory*-Instanz. Die *ManagedConnectionFactory*-Instanz bietet entweder eine existierende Verbindung zu einem EIS, oder erzeugt eine neue physikalische Verbindung zu einem EIS.

Eine *ManagedConnection*-Instanz repräsentiert eine physikalische Verbindung zu einem EIS. Der Applikationsserver übergibt eine Menge von Kandidaten als Parameter für die Methode *matchManagedConnection()*, um festzustellen, ob es eine passende *ManagedConnection*-Instanz für die Anfrage gibt. Diese Methode ist vom Resource-Adapter nach eigenen Kriterien implementiert. Wenn es keine passende Instanz gibt, ruft der Applikationsserver weiter die Methode *createManagedConnection()*, um eine neue Instanz zu erzeugen, die eine neue physikalische Verbindung zu einem EIS repräsentiert.

Die Methode *equals()* und *hashCode()* werden von dem Applikationsserver benutzt, um den Connection-Pool zu strukturieren. Die Methode *ManagedConnectionFactory.equals()* vergleicht zwei Objekte, ob sie identisch sind. Die Methode *ManagedConnectionFactory.hashcode()* liefert die Hashcode eines Objektes zurück. Die beiden Methoden sollen so implementiert werden, dass die *ManagedConnectionFactory*-Instanz eindeutig, und zu einem EIS spezifiziert ist. Basierend auf die beiden Methoden übergibt der Applikationsserver nach eigenen Kriterien die Menge der Kandidaten als Parameter für die Methode *matchManagedConnection()*.

ManagedConnection

Eine Instanz des *javax.resource.spi.ManagedConnection*-Interfaces repräsentiert eine physikalische Verbindung zu einem EIS. Dieses Interface muss von dem Resource-Adapter implementiert werden. Wenn eine Instanz dieses Interfaces erzeugt wird, werden EIS- und Resource-Adapter-Ressourcen(zum Beispiel Speicher und Netzwerk-Socket) für diese physikalische Verbindung allokiert. Um diese Kosten zu verringern, sammelt der Applikationsserver diese *ManagedConnection*-Instanzen im Pool, um diese Instanz wieder verwenden zu können.

Für Connection-Management-Contract muss ein Resource-Adapter die folgenden Methoden vom *ManagedConnection*-Interface implementieren.

```
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;
```

```
  public void addConnectionEventListener(
                                    ConnectionEventListener listener);
  public void removeConnectionEventListener(
                                    ConnectionEventListener listener);
  public ManagedConnectionMetaData getMetaData()
      throws ResourceException;
}
```

Die Methode *ConnectionFactory.getConnection()* auf Resource-Adapter wird von dem Applikationsserver aufgerufen, um eine neue Verbindung auf Applikationsebene zu erzeugen, die mit dieser *ManagedConnection*-Instanz verbunden ist. Diese Verbindung auf Applikationsebene ist von dem Typ *javax.resource.cci.Connection.* Diese Methode wird auch benutzt, um den Zustand einer physikalischen Verbindung zu ändern. Zum Beispiel, kann ein Applikationsserver diese Methode aufrufen, um eine erneute Authentifizierung der physikalischen Verbindung zu dem EIS zu beginnen.

Nachdem eine *ManagedConnection*-Instanz erzeugt wurde, ruft der Applikationsserver die Methode *addConnectionEventListener() auf*, um eine *ConnectionEventListener*-Instanz bei einer *ManagedConnection*-Instanz zu registrieren. Mit diesem Listener kann die *ManagedConnection*-Instanz dem Applikationsserver die Ereignisse für das Beenden von Verbindungen oder Fehler usw. benachrichtigen. Zum Beispiel, wenn eine Verbindung auf Applikationsebene von der Applikation beendet wurde, benutzt die *ManagedConnection*-Instanz die Methode *myListener.connectionclose(ConnectionEvent)* des Listeners *myListener*, um dem Applikationsserver Ereignisinformationen mitzuteilen.

Die Methode *removeConnectionEventListener()* entfernt eine *ConnectionEventListener*-Instanz von einer *ManagedConnection*-Instanz.

Die Methode *ManagedConnection.cleanup()* wird von dem Applikationsserver aufgerufen, um alle Verbindungen auf Applikationsebene, die mit dieser *ManagedConnection*-Instanz verbunden sind, zu beenden. Das passiert, zum Beispiel, wenn der Server einen Connection-Close-Request von der *ManagedConnection*-Instanz bekommen hat. Nachdem der Server diese Methode aufgerufen hat, gibt er diese *ManagedConnection*-Instanz in den Connection-Pool zurück.

Der Applikationsserver ruft die Methode *ManagedConnection.destroy()* auf, um die physikalische Verbindung endgültig zu löschen. Zum Beispiel, wenn die Anzahl der *ManagedConnection*-Instanzen im Pool zu hoch ist, muss der Server diese Methode aufrufen, um einige *ManagedConnection*-Instanzen zu löschen. Beim Aufruf dieser Methode muss der Resource-Adapter die System-Ressourcen, die von dieser *ManagedConnection*-Instanz besetzt sind, wieder freigeben.

Die Methode *ManagedConnection.getMetaData()* liefert eine Instanz von *javax.resource.spi.ManagedConnectionMetaData* zurück.

ManagedConnectionMetaData

Das Interface *javax.resource.spi.ManagedConnectionMetaData* liefert Informationen über eine *ManagedConnection*–Instanz und das angebundene EIS. Diese Informationen enthalten den Produktnamen des EISs, die Produktversion des EISs, Benutzername usw.. Ein Resource-Adapter muss das *ManagedConnectionMetaData*-Interface implementieren.

ConnectionEventListener

Die Connector-Architektur bietet einen Callback-Mechanismus, damit ein Resource-Adapter dem Applikationsserver die Meldungen für verschiedene Ereignisse schicken kann. Und der Applikationsserver benutzt diese Meldungen, um den Connection-Pool und lokale Transaktionen zu verwalten.

Das Interface *javax.resource.spi.ConnectionEventListener* ist von dem Applikationsserver zu implementieren.

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

    public void localTransactionStarted(ConnectionEvent event);
    public void localTransactionCommitted(ConnectionEvent event);
    public void localTransactionRolledback(ConnectionEvent event);
}
```

Wenn eine Applikationskomponente eine Verbindung auf Applikationsebene beendet, ruft die *ManagedConnection*-Instanz die Methode *ConnectionEventListener.connectionClosed()* auf, um alle Listener zu benachrichtigen, die an diese Instanz registriert wurden.

Die *ManagedConnection*-Instanz ruft die Methode *ConnectionEventListener.connectionErrorOccurred()* auf, um die Listener mitzuteilen, dass ein Fehler bei der physikalischen Verbindung auftritt.

Die Methoden *localTransactionStarted()*, *localTransactionCommitted()* und *localTransactionRollback()* werden verwendet, um die Ereignisse über die lokalen Transaktionen zu melden.

ConnectionEvent

Eine Instanz der Klasse *javax.resource.spi.ConnectionEvent* enthält die folgenden Informationen:

- Den Typ der Ereignisbenachrichtigung, zum Beispiel CONNECTION_CLOSED;
- Die *ManagedConnection*-Instanz, von der dieses Connection-Event erzeugt wurde;
- Die *Connection*-Instanz, die mit dieser *ManagedConnection*-Instanz verbunden ist, und das ConnectionEvent angefordert hat.

- Optional noch ein *Exception*, das von CONNECTION_ERROR_OCCURRED benutzt wird.

5.2 Common Client Interface(CCI)

Das CCI definiert die Interfaces, die in der Abbildung 5.2[CONSPE10] gezeigt werden. Die Interfaces *javax.resource.cci.ConnectionFactory*, *javax.resource.cci.Connection* und *javax.resource.cci.ConnectionSpec* sind verbindungsbezogen. Sie bieten den Applikationskomponenten an, eine Verbindung zu einem EIS zu erzeugen. Die interaktionsbezogenen Interfaces sind *javax.resource.cci.Interaction* und *javax.resource.cci.InteractionSpec*. Sie ermöglichen der Applikationskomponente, durch eine Interaktion mit einem EIS zu kommunizieren. Um eine Datenstruktur zu repräsentieren, bietet das CCI die Interfaces *javax.resource.cci.Record*, *javax.resource.cci.MappedRecord*, *javax.resource.cci.IndexedRecord*, und *javax.resource.cci.ResultSet*. Das Interface *javax.resource.cci.RecordFactory* bietet zwei Methoden *createMappedRecord()* und *createIndexedRecord()*, um die Instanzen von *MappedRecord* und *IndexedRecord* zu erzeugen.

ConnectionFactory

Eine *ConnectionFactory*-Instanz wird von einer *ManagedConnectionFactory*–Instanz erzeugt, und mit einem JNDI-Namen verbunden. Eine Applikationskomponente sucht eine *ConnectionFactory*-Instanz im JNDI-Namespace, und verwendet diese Instanz, um eine Verbindung zu einer EIS zu bekommen. Die folgende Code zeigt, welche Methoden im *ConnectionFactory*-Interface implementiert werden sollen.

```
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;
}
```

Die Applikationskomponente ruft die Methode *ConnectionFactory.getConnection()* auf, um eine Verbindung auf Applikationsebene zu bekommen, die mit einer physikalischen Verbindung verbunden ist. Um eine physikalische Verbindung zu erzeugen, muss zuerst ein EIS-Sign-On durchgeführt werden. Wenn die Applikationskomponente der Methode keine Parameter übergibt, ist der Server für das EIS-Sign-On zuständig. Die Komponente kann auch der Methode die Sicherheitsinformationen als Parameter übergeben(z.B. username, passwort), um sich beim EIS einzuloggen.

Die Methode *ConnectionFactory.getRecordFactory()* liefert eine *RecordFactory*-Instanz zurück. Mit dieser Instanz können die Instanzen von *IndexedRecord* und *MappedRecord* erzeugt werden.

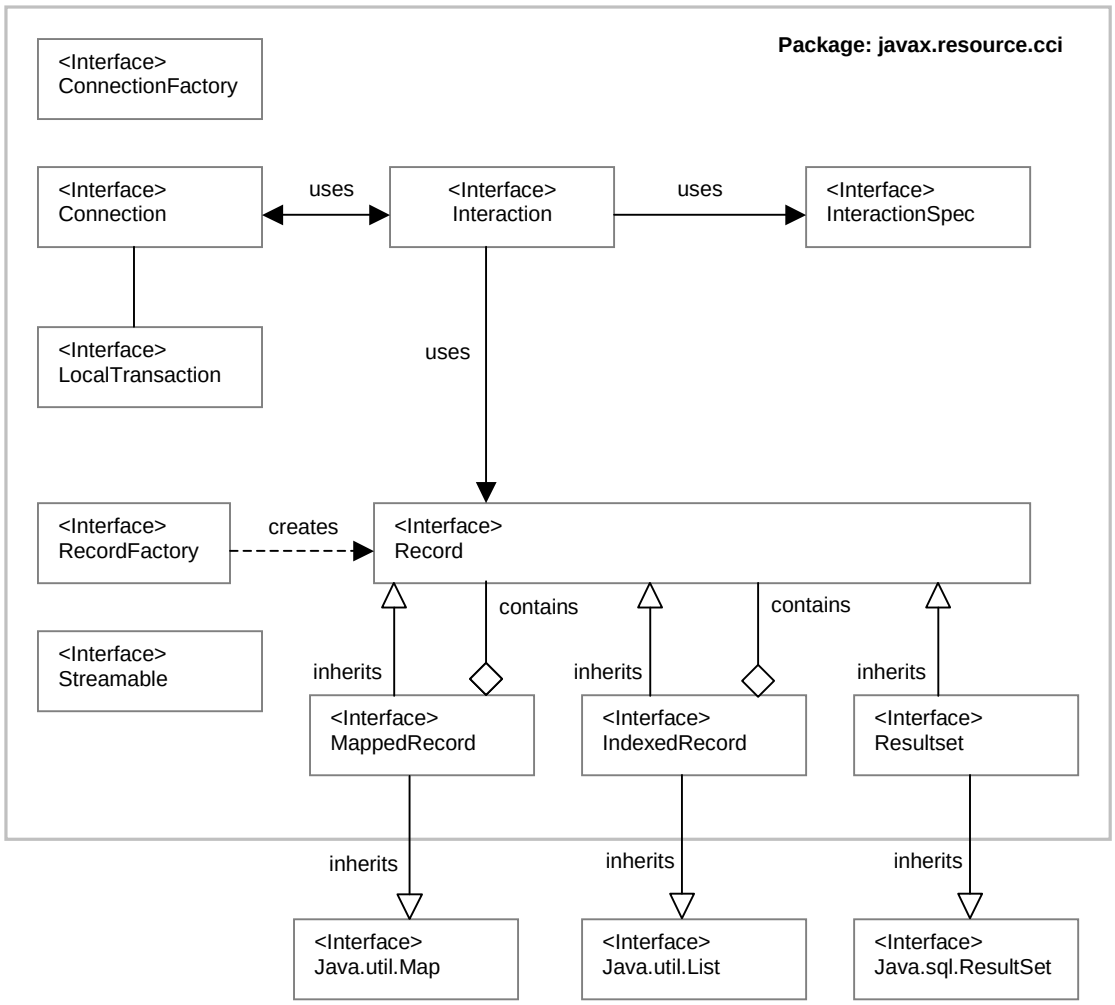


Abbildung 5.2[CONSPE10]: Common Client Interface

ConnectionSpec

Das Interface *javax.resource.cci.ConnectionSpec* wird von einer Applikationskomponente benutzt, um die Sicherheitsinformationen und den Connection-Parameter zu übertragen. Eine Instanz von diesem Interface wird von der Methode *getConnection()* im Interface *ConnectionFactory* als Parameter benutzt.

Connection

Eine Instanz von dem Interface *javax.resource.cci.Connection* repräsentiert eine Verbindung auf Applikationsebene, die mit einer *ManagedConnection*-Instanz verbunden ist. Diese Instanz wird von der Applikationskomponente verwendet, um mit dem EIS zu kommunizieren.

Nachdem die Applikationskomponente eine Instanz von *Conection* bekommen hat, ruft die Komponente die Methode *Connection.createInteraction()* auf, und bekommt eine *Interaction*-Instanz.

Das Interface *Connection* bietet noch zusätzlich die Methode *close()* für die Komponente. Um die Verbindung auf Applikationsebene zu beenden, muss die Komponente diese Methode aufrufen.

Interaction

Das Interface *javax.resource.cci.Interaction* ermöglicht einer Komponente, die Funktionen auf einem EIS auszuführen. Um das zu realisieren, bietet es die Methode *Interaction.execute()* mit den Parametern input-*Record* und *InteractionSpec*. Diese Methode liefert ein output-*Record* als Rückgabewert zurück. Record repräsentiert eine Datenstruktur wie zum Beispiel *IndexedRecord* oder *MappedRecord*. Das wird jedoch später noch einmal näher erklärt.

Eine *javax.resource.cci.InteractionSpec*-Instanz enthält die Properties, die für die Ausführung einer Interaktion notwendig sind. Das Interface soll so implementiert werden, dass es die getter- und setter-Methoden für jede Property anbietet. Zum Beispiel kann, um eine Funktion im EIS auszuführen, der jeweilige Funktionsname hier im InteractionSpec mit getter- und setter-Methoden definiert werden.

Metadata

Das Interface *javax.resource.cci.ConnectionMetaData* bietet die Informationen über eine EIS-Instanz, die mit einer *Connection*-Instanz verbunden ist. Eine Komponente ruft die Methode *connection.getMetaData()* auf, um eine *ConnectionMetaData*–Instanz zu bekommen.

Das Interface *javax.resource.cci.ResourceAdapterMetaData* bietet Informationen über eine ResourceAdapter-Implementation. Eine Komponente benutzt die Methode *ConnectionFactory.getMetaData()* um die Meta-Daten-Informationen über ein Resource-Adapter zu bekommen.

Der Unterschied zwischen *ConnectionMetaData* und *ResourceAdapterMetaData* liegt in folgenden Punkten. Erstens, *ConnectionMetaData* bietet die Informationen über ein EIS, während *ResourceAdapterMetaData* die Informationen über den Resource-Adapter liefert. Zweitens, *ConnectionMetaData* ist mit einer aktiven Connection verbunden, während *ResourceAdapterMetaData* mit einer ConnectionFactory verbunden ist. Das heißtt, um die *ConnectionMetaData* zu bekommen, muss eine aktive Connection vorhanden sein. Aber es muss keine aktive Verbindung zu einem EIS angefordert werden, um die *ResourceAdapterMetaData* zu bekommen.

ConnectionMetaData unterscheidet sich auch von *ManagedConnectionMetaData*. *ManagedConnectionMetaData* liefert nicht nur die Informationen über ein EIS, sondern auch die Informationen über die physikalische Verbindung, nämlich der *ManagedConnection*-Instanz. Um diese Informationen zu bekommen, muss die physikalische Verbindung gültig sein.

Exception Interfaces

Das Interface *javax.resource.cci.ResourceException* ist als die Wurzel der Exception-Hierarchie für das CCI anzusehen.

Das Interface *javax.resource.cci.ResourceWarning* bietet die Warnungsinformationen, die sich auf eine Interaktion beziehen. Eine Komponente ruft die Methode *Interaction.getWarning()* auf, um auf die erste ResourceWarning der erzeugten Warnungen zuzugreifen.

Record

Ein Record ist eine Java-Repräsentation von einer Datenstruktur, die als Input oder Output für eine EIS-Funktion benutzt wird. Interface *javax.resource.cci.Record* ist das Basisinterface für die Repräsentation von einem Record. Die Methode *Interaction.execute()* verwendet die Record-Instanz als Parameter.

Das Record-Interface kann erweitert werden, um eine der folgenden Repräsentation zu erzeugen: *javax.resource.cci.MappedRecord, javax.resource.cci.IndexedRecord* und *javax.resource.cci.Resultset*.

Das *MappedRecord* Interface kann für die Repräsentation einer Sammlung von Record-Elementen als Schlüssel-Werte Paar verwendet werden. Das Interface erweitert die Interfaces *Record* und *java.util.Map*. Ein Objekt bildet einen Schlüssel auf einen Wert ab, wobei jeder Schlüssel höchstens auf genau einen Wert abgebildet werden kann. Zum Beispiel, mit *WorkflowMappedReocord.put("name", "John Smith")* wird der Schlüssel "name" auf den Wert "John Smith" abgebildet und das Objekt zu einer *MappedRecord*-Instanz hinzugefügt.

Das *IndexedRecord* Interface repräsentiert eine geordnete Folge von Elementen, die auf *java.util.List* basiert. Der Benutzer kann auf die Elemente über einen Index zugreifen oder Elemente hinzufügen.

Um die Instanzen von *MappedRecord* oder *IndexedRecord* zu erzeugen, wird das Interface *RecordFactory* verwendet.

Das *ResultSet*-Interface erweitert die Interfaces *Record* und *java.sql.ResultSet*. Ein ResultSet repräsentiert diejenigen Daten, die von einem EIS durch die Ausführung einer Interaktion erhalten werden, in tabellarischer Form.

Das Interface *javax.resource.cci.Streamable* ermöglicht einem ResourceAdapter die Daten vom Input-Record zu bekommen, und die Daten als Bitstrom in ein Output-Record zu setzen. Dieses Interface wird nicht von einer Komponente direkt verwendet, sondern von der Implementierung eines Resource-Adapters.

Verwendung des CCIs

Eine Applikationskomponente kann durch CCI eine Verbindung zu einem EIS erstellen, auf das EIS zugreifen, die Funktionen auf das EIS ausführen, und schließlich die Verbindung beenden. Die grundlegenden Schritte bei der Programmierung sind wie folgt:

- Erstellen eines JNDI-Contextes für die Sitzung;

  z.B. `Context ic = new InitialContext();`

- Benutzen der Methode *lookup()*, um eine *ConnectionFactory*-Instanz zu finden;

  z.B.
  ```
  ConnectionFactory cf
          = (ConnectionFactory) ic.lookup("java:comp/env/MYADAPTER");
  ```

  Hier ist MYADAPTER der JNDI-Name der *ConnectionFactory*-Instanz.

- Benutzen der Methode *ConnectionFactory.getConnection()*, um eine Verbindung zu einem EIS zu erzeugen;

  z.B.
  ```
  Connection con  =  cf.getConnection();
  ```

- Benutzen der Methode *Connection.createInteraction()*, um eine *Interaction*-Instanz zu erzeugen;

  z.B.
  ```
  Interaction ix = con.createInteraction();
  ```

- Erzeugen einer *InteractionSpec*-Instanz;

  z.B.
  ```
  MyInteractionSpec iSpec = new MyInteractionSpec();
     iSpec.setFunctionName(“name”);
  ```

- Benutzen der Methode *ConnectionFactory.getRecordFactory()*, um eine Referenz einer *RecordFactory*-Instanz zu erhalten;

  z.B.
  ```
  RecordFactory rf = cf.getRecordFactory();
  ```

- Erzeugen einer *Record*-Instanz durch den Methoden-Aufruf einer *RecordFactory*-Instanz. Zum Beispiel, um eine *IndexedRecord*-Instanz zu erzeugen, muss die Methode *createIndexedRecord()* aufgerufen werden:

  ```
  IndexedRecord iRec = rf.createIndexedRecord();
  iRec.add(obj);
  ```

- Aufrufen der Methode *Interaction.execute()* mit Parameter *Input*-Record, und *InteractionSpec*, um die Funktionen auf dem EIS auszuführen, und das *Output*-Record zu erhalten;

  z.B.
  ```
  Record oRec = ix.execute( iSpec, iRec );
  Iterator iterator = ((IndexedRecord)oRec).iterator();
  while ( iterator.hasNext() )
  {
    Object obj = iterator.next();
    <Datenauslesen vom Objekt>
  }
  ```

- Beenden der Verbindung zum EIS:
  `con.close();`

5.3 Security Management Interfaces

In der Connector-Architektur wird ein Security-Contract zwischen dem Applikationsserver und dem EIS definiert. Dieser Kontrakt gewährleistet die Sicherheit eines EISs vor dem Zugriff von der Applikationskomponente. Um auf ein EIS zugreifen zu können, muss zuerst ein EIS-Sign-On durchgeführt werden.

Für eine Applikationskomponente gibt es zwei Möglichkeiten, um ein EIS-Sign-On durchzuführen. Die erste Möglichkeit ist "Container-managed Sign-on". In diesem Fall nimmt der Applikationsserver die Verantwortung für das EIS-Sign-On. Die zweite Möglichkeit ist "Component-Managed Sign-on". Die Komponente verwaltet selbst das EIS-Sign-On.

Der Security-Contract zwischen einem Applikationsserver und einem Resource-Adapter basiert auf dem Connection-Management-Contract. Um EIS-Sign-On zu unterstützen, werden die folgenden Klassen und Interfaces verwendet.

Subject

Ein *Subject* repräsentiert die Informationen eines Entity's, zum Beispiel einer Person. Diese Informationen enthalten die Subject-Identitäten, sogenannte *Principals*, und die Security-Attribute, auch als *Credential* bezeichnet. Ein *Principal* bindet einen Namen an ein *Subject*. Ein Subject wird von dem Applikationsserver zu dem Resource-Adapter übertragen, um es zu authentifizieren und autorisieren.

Generic Credential

Das Interface *javax.resource.spi.security.GenericCredential* ist unabhängig von Sicherheitsmechanismen. Es wird benutzt, um das sicherheitsmechanismus-spezifische-Security-Credential einzupacken, zum Beispiel, Kerberos-Credential.

Wenn der Resource-Adapter das Interface als ein Teil des Security-Contracts unterstützt, dann muss der Applikationsserver, worauf der Resource-Adapter läuft, das Interface auch unterstützen.

PasswordCredential

Die Klasse *javax.resource.spi.security.PasswordCredential* ist mit einem Behälter von Benutzername und Passwort Einträgen zu vergleichen. Diese Klasse ermöglicht dem Applikationsserver, den Benutzernamen und das Passwort an einen Resource-Adapter zu übertragen.

ConnectionFactory

Wenn die Applikationskomponente die Methode *ConnectionFactory.getConnection()* aufruft, fragt die *ConnectionFactory*-Instanz den Applikationsserver durch die Methode *ConnectionManager.allocateConnection()* an, um eine Verbindung zu lokalisieren.

Im Fall "Container-managed Sign-on" wird die Komponente keine Sicherheitsinformationen zu dem Resource-Adapter übertragen. Dann übergibt die *ConnectionFactory*-Instanz auch keine Sicherheitsinformationen an den Applikationsserver. Die Sicherheitsinformationen werden bei der Konfiguration des Resource-Adapters von dem Deployer eingegeben.

Im Fall "Component-Managed Sign-on" übergibt die Komponente die Sicherheitsinformationen (z.B. Benutzername und Passwort) als Parameter *ConnectionSpec* für die Methode *ConnectionFactory.getConnection()*. Diese Informationen werden vom Resource-Adapter zu dem Applikationsserver weitergeleitet. Die Informationen werden zuerst aus der *ConnectionSpec*-Instanz gelesen und anschliessend in einer *ConnectionRequestInfo*-Instanz eingepackt. Diese Instanz wird dann als Parameter der Methode *ConnectionManager.allocateConnection()* vom Applikationsserver zu dem Resource-Adapter übertragen.

Es gibt einen Unterschied zwischen dem Interface *ConnectionSpec* und *ConnectionRequestInfo*. Das Interface *ConnectionSpec* ist im CCI definiert, und wird auf Applikationsebene verwendet, während *ConnectionRequestInfo* im System-Contract definiert ist, und auf der Serverebene verwendet wird. Das ermöglicht eine saubere Trennung zwischen Systemebene und Komponentenebene.

ManagedConnectionFactory

Im *ManagedConnectionFactory*-Interface wird die Methode *createManagedConnection()* wie folgt definiert:

```
public interface javax.resource.spi.ManagedConnectionFactory
      extends java.io.Serializable
{
  public ManagedConnection createManagedConnection(
                javax.security.auth.Subject subject,
                ConnectionRequestInfo cxRequestInfo)
    throws ResourceException;
}
```

Im Fall "Container-Managed Sign-on" wird die Sicherheitsinformationen als Parameter *Subject* vom Server zum Resource-Adapter übertragen. Die *ManagedConnectionFactory*-Instanz benutzt diese Information, um eine physikalische Verbindung zu dem EIS zu erzeugen.

Im Fall "Component-Managed Sign-on" übergibt der Server *null* für den Parameter *Subject*, und auch die von der Komponente übergebenen Informationen, die in der *ConnectionRequestInfo*-Instanz eingepackt sind, an den Resource-Adapter. Falls die

ConnectionRequestInfo-Instanz null ist, wird die Default-Sicherheitsinformation von der *ManagedConnectionFactory*-Instanz verwendet. Sonst werden die Informationen von der *ConnectionRequestInfo*-Instanz ausgepackt und weiter verwendet.

ManagedConnection

Ein Resource-Adapter kann eine physikalische Verbindung, die im Connection-Pool schon existiert ist, re-authentifizieren. Das heißt, ein Resource-Adapter kann die Sicherheitsinformationen für diese Verbindung ändern.

Der Resource-Adapter wird die physikalische Verbindung re-authentifizieren, wenn der Applikationsserver die Methode *getConnection()* mit dem Parameter *Subject* aufruft, das sich von dem zuvor mit der physikalischen Verbindung assoziierten *Subject* unterscheidet. Wenn die Parameter *Subject null* ist, wird die Sicherheitsinformationen von *ConnectionRequestInfo* verwendet. Ob ein Resource-Adapter Re-Authentifizierung unterstützt, hängt davon ab, ob das verwendete EIS die Re-Authentifizierung für die existierende Verbindung unterstützt. Wenn der Resource-Adapter die Re-Authentifizierung nicht unterstützt, wird die *getConnection*-Methode die Exception *javax.resource.spi.SecurityException* werfen, falls die Sicherheitsinformationen geändert wurden.

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

}
```

5.4 Transaction Management Interfaces

Die Transaction-Management-Interfaces ermöglichen den Applikationsserver, die Transaktionen mit dem EIS für die Applikationen zu verwalten und auszuführen. Eine Applikation kann den transaktionalen Zugriff über mehrere verteilte EIS durchführen. Falls es nur um ein EIS geht, werden die Transaktion von diesem EIS intern verwaltet. Dies wird als lokale Transaktion bezeichnet. Wenn die Transaktion über mehrere EISs geht, ist ein externer Transaction-Manager für die Transaktionsverwaltung zuständig. Diese Art von Transaktion heißt XA-Transaktion. Die Transaktionen werden über mehrere EISs durchgeführt.

Die Connector-Architektur unterstützt sowohl XA-Transaction (or JTA-Transaction) als auch Local-Transaction. Ein Resource-Adapter kann entweder beide Transaktionen unterstützen, nur eine davon, oder gar keine.

Die Connector-Architektur definiert den Transaction-Management-Contract zwischen dem Applikationsserver und dem Resource-Adapter(und darunterliegende Resource-Manager). Der Transaction-Management-Contract enthält zwei Teile. Der erste Teil ist der *javax.transaction.xa.XAResource* basierte Kontrakt zwischen einem Transaction-Manager und einem Resource-Manager für die XA-Transaction. Der zweite Teil ist der Local-Transaction-

Management-Kontract zwischen einem Applikationsserver und einem Resource-Manager für die lokale Transaktionen, dieser Kontrakt basiert auf dem Interface *javax.resource.spi.LocalTransaction*. Für die lokale Transaktion definiert die Connector-Architektur noch das Interface *javax.resource.cci.LocalTransaction* für die Applikationskomponenten.

ManagedConnection

Das *ManagedConnection*-Interface bietet die Möglichkeit, auf zwei Interfaces zuzugreifen: *javax.transaction.xa.XAResource* und *javax.resource.spi.LocalTransaction*. Das Interface *XAResource* wird von dem Transaction-Manager benutzt, um eine Transaktion mit dem Resource-Manager des EISs zu assoziieren oder von ihm zu trennen. Der Applikationsserver benutzt das Interface *javax.resource.spi.LocalTransaction*, um die lokalen Transaktionen zu verwalten.

Das *ManagedConnection*-Interface definiert die Methoden *getXAResource()* und *getLocalTransaction()*. Die beiden Methoden werden von dem Applikationsserver aufgerufen, und liefern jeweils die Instanz von *XAResource* oder *javax.resource.spi.LocalTransaction* zurück.

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

XAResource

Das Interface *XAResource* definiert den Vertrag zwischen Resource-Adapter und einem Transaction-Manager für verteilte Transaktionen. Dieses Interface ermöglicht den Resource-Manager eines EISs an verteilten Transaktionen teilzunehmen, die von einem externen Transaction-Manager kontrolliert und koordiniert werden. Der Transaction-Manager benutzt die *XAReosurce*-Instanz, um mit dem Resource-Manager zu kommunizieren, um die Transaktion zu assoziieren, zu beenden oder wiederherzustellen. Das *XAReource*-Interface muss von einem Resource-Adapter implementiert werden, wenn der Resource-Adapter die XA-Transaktion unterstützt. In diesem Fall muss die 1:1 Beziehung zwischen der *ManagedConnection*-Instanz und der *XAResource*-Instanz beibehalten werden. Das heißt, jedes Mal wenn die Methode *ManagedConnection.getXAResource()* aufgerufen wird, muss die selbe *XAResource*-Instanz zurückgeliefert werden.

```
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;
}
```

Falls ein Resource-Adapter und sein Resource-Manager die XA-Transaktion unterstützt, dann muss die Methode *XAResource.commit(onePhase = true)* implementiert werden, um das *one-phase-commit*-Protokoll zu unterstützen. Sie können auch die *two-phase-commit*-Protokoll oder die Kombination von der *one-phase-commit*-Protokoll und der *two-phase-commit*-Protokoll unterstützen.

Wenn eine Applikationskomponente eine Verbindung zu einem EIS von dem Resource-Adapter verlangt, wird eine physikalische Verbindung(*ManagedConnection*-Instanz) entweder vom Connection-Pool auf dem Applikationsserver wiederhergestellt oder neu erzeugt. Danach wird der Applikationsserver die Methode *ManagedConnection.getXAResource()* aufrufen, um eine *XAResource*-Instanz zu bekommen. Dann assoziiert der Applikationsserver diese *XAResource*-Instanz mit dem Transaction-Manager, um dem Transaction-Manager mitzuteilen, dass ein Resource-Manager an die aktuelle Transaktion teilnimmt. Der Transaction-Manager ruft die Methode *XAResource.start(),* um die *XAResource*-Instanz mit der aktuellen Transaktion zu assoziieren.

Wenn eine Applikationskomponente die Verbindung zu dem EIS beendet, schickt der Resource-Adapter dem Applikationsserver die Benachrichtigung (CONNECTION_CLOSED). Der Applikationsserver wird zuerst die *XAResource*-Instanz, die mit dieser physikalischen Verbindung verbunden ist, von der Transaktion trennen, bevor er *cleanup* von der physikalischen Verbindung durchführt. Der Transaction-Manager ruft die Methode *XAResource.end()* auf, um dem Resource-Manager mitzuteilen, dass die *ManagedConnection*-Instanz, die mit dieser *XAResource*-Instanz 1:1 verbunden ist, von der Transaktion deassoziiert wird.

Falls mehrere Resource-Manager an einer Transaktion teilnehmen, wird der Transaction-Manager warten, bis alle *XAResource*-Instanzen von ihm getrennt wurden. Anschliessend startet er einen Prozess auf Basis des two-phase-commit-Protokoll. Zuerst ruft der Transaction-Manager die Methode *XAResource.prepare()* auf dem Resource-Manager auf. Falls jeder Resource-Manager, der an dieser Transaktion teilgenommen hat, zustimmt, ruft der Transaction-Manager die Methode *XAResource.commit()* bei jedem Resource-Manager auf, um die Transaktion zu committen. Sonst ruft er die Methode *XAResource.rollback()* auf, um die Transaktion zurückzusetzen.

Falls nur ein Resource-Manager an einer Transaktion teilgenommen hat, wird *one-phase-commit* durchgeführt. Das heißtt, der Transaction-Manager ruft nicht zuerst die Methode *XAResource.prepare()* auf dem Resource-Manager auf, sondern direkt die Methode *XAResource.commit()* auf, um die Transaktion zu committen. Falls der Resource-Manager daran scheitert, die Transaktion zu committen, dann setzt er die Transaktion zurück.

javax.resource.spi.LocalTransaction

Das *LocalTransaction*-Interface definiert einen Kontrakt zwischen einem Applikationsserver und einem Resource-Manager. Der Applikationsserver benutzt dieses Interface, um die lokale

Transaktionen zu verwalten. Ein Resource-Adapter implementiert das Interface, um die lokale Transaktion, die auf dem darunterliegenden Reosource-Manager ausgeführt wird, zu unterstützen.

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

javax.resource.cci.Connection

Das Interface *Connection* bietet den Applikationskomponenten die Möglichkeit, eine Instanz von *javax.resource.cci.LocalTransaction* zu bekommen, damit die Applikationskomponenten die Transaktion auf Applikationsebene verwalten können.

```
public interface javax.resource.cci.Connection
{
  public javax.resource.cci.LocalTransaction getLocalTransaction()
      throws ResourceException;
  //...
}
```

javax.resource.cci.LocalTransaction

Der Unterschied zwischen dem Interface *javax.resource.spi.LocalTransaction* und *javax.resource.cci.LocalTransaction* liegt darin, dass das Interface *javax.resource.spi.LocalTransaction* als ein Teil vom System-Contract zwischen dem Applikationsserver und dem Resource-Adapter definiert wird, während *javax.resource.cci.LocalTransaction* ein Interface auf Applikationsebene zwischen den Applikationskomponenten und einem Resource-Manager definiert wird. Das Interface *javax.resource.spi.LocalTransaction* wird von dem Container im Applikationsserver verwendet, während das Interface *javax.resource.cci.LocalTransaction* von den Applikationskomponenten verwendet wird.

Für die Applikationskomponente gibt es zwei Arten von Transaktionen: Container-Managed-Transaction und Component-Managed-Transaction. Im Fall von Container-Managed-Transaction verwaltet ein Container die Transaktionen, während im Fall von Component-Managed-Transaction die Applikationskomponenten für die Transaktionen zuständig sind. Das Interface *javax.resource.cci.LocalTransaction* bietet den Applikationskomponenten die Möglichkeit, die lokale Transaktionen selbst zu verwalten. Ein Resource-Adapter muss dieses Interface aber nicht implementieren.

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

Eine Applikationskomponente ruft die Methoden *LocalTransaction.begin()*, *LocalTransaction.commit()* und *LocalTransaction.rollback()* auf, um eine Transaktion zu starten, committen oder zurückzusetzen. Wenn eine Applikationskomponente diese Methode aufruft, dann muss der Resource-Adapter dem Applikationsserver über diese Ereignisse benachrichtigen. Das Interface *javax.resource.spi.ConnectionEventListener* bietet dem Resource-Adapter diese Möglichkeit.

Der Applikationsserver benutzt die Nachrichten, die der Resource-Adapter ihm geschickt hat, um die Transaktionen zu verwalten. Zum Beispiel startet ein Session-Bean eine Transaktion mit *LocalTransaction.begin()* innerhalb eines Methodeaufrufs, dann schickt der Resource-Adapter dem Applikationsserver die Nachricht über das Ereignis. Wenn die Methode zurückkehrt, ohne die Transaktion zu beenden oder zurückzusetzen, wird der Applikationsserver die Session-Bean-Instanz terminieren und dem Client eine Exception werfen.

javax.resource.spi.ConnectionEventListener

Ein Applikationsserver implementiert das Interface *ConnectionEventListener*. Der Applikationsserver registriert den Listener an einer *ManagedConnection*-Instanz im Resource-Adapter, damit der Resource-Adapter dem Applikationsserver über die Ereignisse benachrichtigen kann. Für die lokalen Transaktionen wurden drei Methoden im Interface *ConnectionEventListener* definiert: *localTransactionStarted(), localTransactionCommitted()* und *localTransactionRolledback()*. Mit den Methoden schickt der Resource-Adapter dem Applikationsserver die Meldungen, dass eine Transaktion gestartet, committed oder zurückgesetzt wurde. Zum Beispiel, ruft der Resource-Adapter die Methode *ConnectionEventListener.localTransactionStarted (LOCAL_TRANSACTION_STARTED)* auf, um dem Applikationserver mitzuteilen, dass eine lokale Transaktion schon gestarted wurde. Im Fall von Container-Managed-Transaction muss der Resource-Adapter dem Applikationsserver nicht benachrichtigen.

```
public interface javax.resource.spi.ConnectionEventListener
{
  public void localTransactionStarted(ConnectionEvent event);
  public void localTransactionCommitted(ConnectionEvent event);
  public void localTransactionRolledback(ConnectionEvent event);
  //...
}
```

6 Realisierung des Resource-Adapters für MQSeries Workflow

In dieser Diplomarbeit wird der Resource-Adapter für MQSeries-Workflow nach dem Standard der “j2EE Connector Architecture Specification, Final release 1.0” implementiert. Hierzu werden der IBM WebSphere Application Server 4.0 Advance Edition (WAS), und IBM MQSeries Workflow 3.3 (MQWF) verwendet.

Die WAS-Implementation basiert auf der J2EE Connector Architecture Specification, Proposed Final Draft #2 Version[WASINFO, Kapitel 4, Abschnitt 4.9.1], der nur "component-managed Sign-On" unterstützt[WASINFO, Kapitel 4, Abschnitt 4.9.1]. Für den Authentifizierungsmechanismus unterstützt er nur *PasswordCredential* als *Credential*-Interface, und BasicPassword als Authentifizierungsmechanismus. Die Verwendung von *GenericCredential* oder *Kerbv5* wird von dem WAS ignoriert[WASINFO, Kapitel 4, Abschnitt 4.9.1].

Der Resource-Adapter verwendet die MQ Workflow-Java-API, um mit dem MQWF-Server zu kommunizieren.

Die J2EE-Connector-Architektur definiert die Interfaces, die den Connection-Management-Contract, den Transaction-Management-Contract und den Security-Contract unterstützen. In Kapitel 4 wurden diese Interfaces schon genau beschrieben. Basierend auf dieser Beschreibung werden hier die wichtigen Methoden, die im Resource-Adapter für MQWF implementiert werden, noch genau erklärt.

Ein Resource-Adapter wurde so implementiert, dass er Managed- und Non-Managed-Applikationsszenario unterstützt. In diesem Kapitel wird beschrieben, wie ein Reosurce-Adapter im Managed-Applikationsszenario implementiert und verwendet wird. In Kapitel 7 wird erklärt, wie ein Applikationsclient einen Resource-Adapter im Non-Managed-Applikationsszenario verwendet.

6.1 Aufbau einer Verbindung

Eine Applikationskomponente fordert eine Verbindung zu einem MQWF-Server von dem Resource-Adapter an. Der Resource-Adapter erzeugt eine physikalische Verbindung zu dem MQWF-Server, und liefert der Applikationskomponente eine Verbindung auf Applikationsebene zurück, die mit der physikalischen Verbindung assoziiert ist. Mit dieser Verbindung kann die Komponente auf den MQWF-Server zugreifen. Die Abbildung 6.1 stellt ein Szenario dar, wie eine neue Verbindung zu dem MQWF-Server erzeugt wird.

Beim Installieren eines Resource-Adapters wird eine Instanz von einer *ConnectionFactory* erzeugt, wobei diese Instanz mit einem JNDI-Name verbunden wird. Eine Applikationskomponente (z.B. ein EJB) sucht die *ConnectionFactory*-Instanz im JNDI-Namespace, und ruft die Methode *ConnectionFactory.getConnection()* im Resource-Adapter auf.

```
Context ic = new InitialContext();
cf = (ConnectionFactory) ic.lookup("java:comp/env/MYADAPTER");
ConnectionSpec spec = new MyConnectionSpec(user, password);
Connection con = cf.getConnection(spec);
```

Hier werden der Benutzername und das Passwort, die zum Einloggen auf dem MQWF-Server benutzt werden, in einer *ConnectionSpec*-Instanz eingepackt, und als Parameter für die Methode *ConnectionFactory.getConnection()* verwendet.

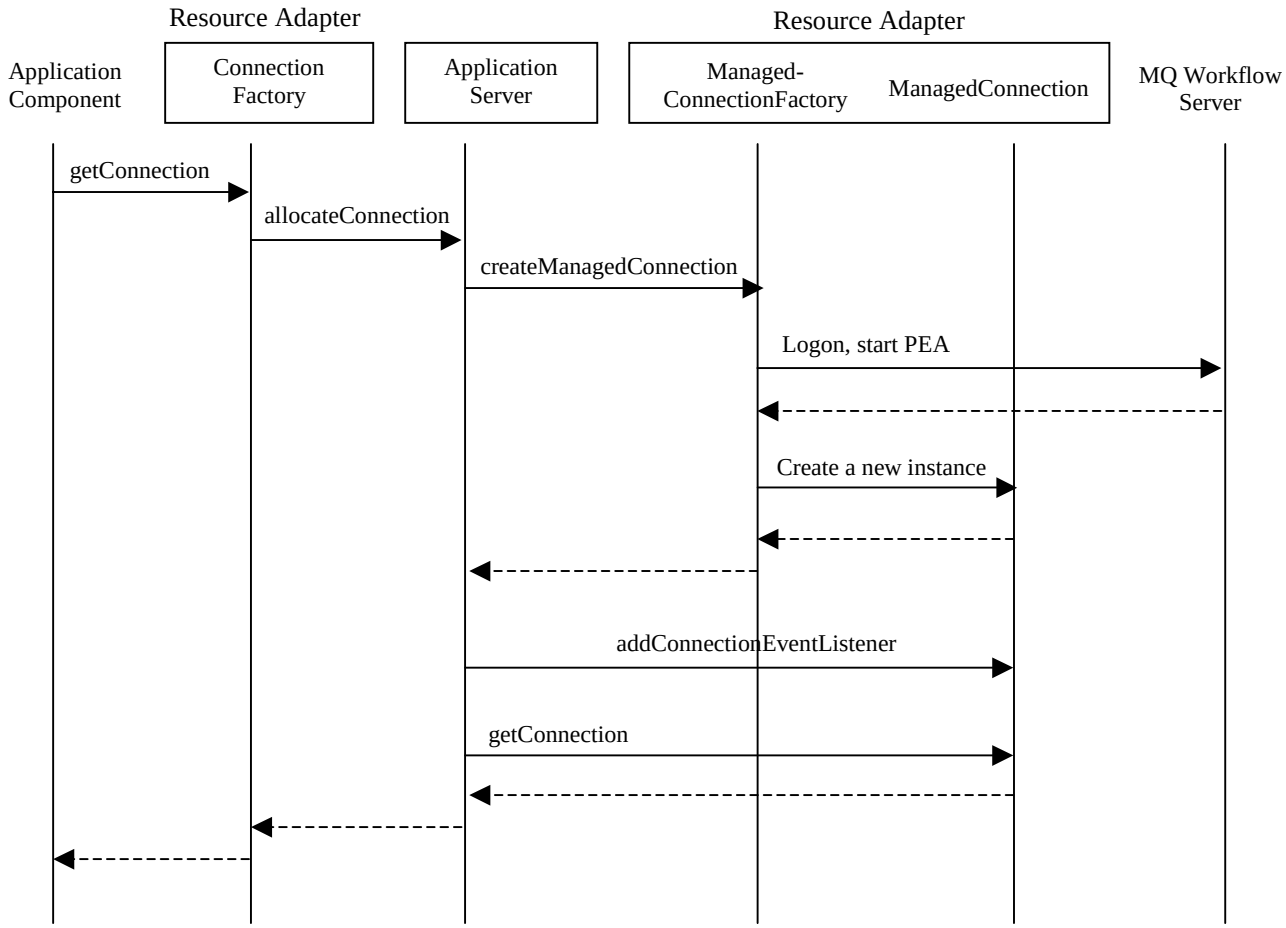


Abbildung 6.1[CONSPE10]: Neue Verbindungserzeugung im Managed-Applikationsszenario

Nachdem die *ConnectionFactory*-Instanz im Resource-Adapter die Verbindungserzeugungsanfrage von der Applikationskomponente erhalten hat, ruft sie die Methode *ConnectionManager.allocateConnection()* auf dem Applikationsserver auf.

```
public Connection getConnection(ConnectionSpec properties)
throws ResourceException
{
  ConnectionRequestInfo info = new MyConnectionRequestInfo(
                            (MyConnectionSpec)properties).getUser(),
                            ((MyConnectionSpec)properties).getPassword());
  Connection con = (Connection) cm.allocateConnection(mcf,info);
```

```
  return con;
}
```

Hier wird der Benutzername und das Passwort von der *ConnectionSpec*-Instanz ausgelesen und wieder in einer *ConnectionRequestInfo*-Instanz eingepackt. Diese Instanz wird dann als ein Parameter für die Methode *ConnectionManager.allocateConnection()* vom Resource-Adapter an den Applikationsserver übergegeben.

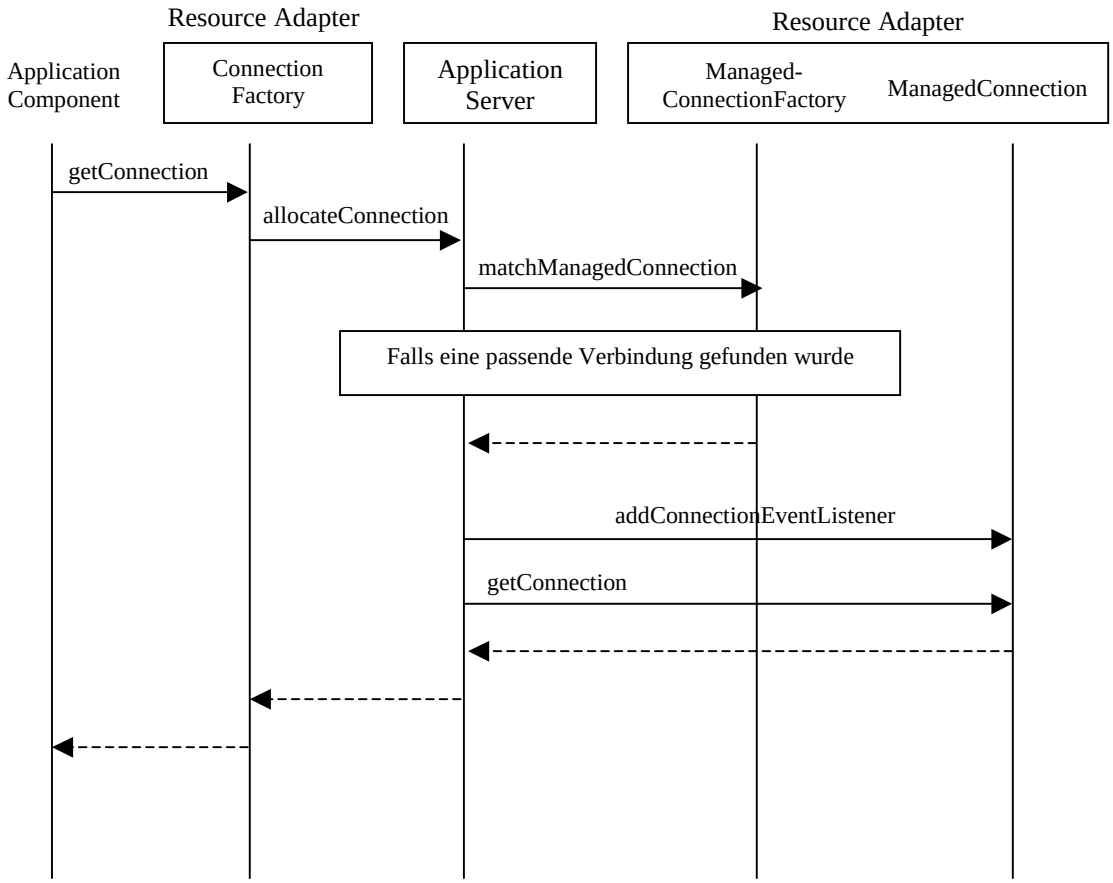


Abbildung 6.2[CONSPE10]: Wiederverwendung von Verbindungen im Managed-Applikationsszenario

Falls der Applikationsserver die Kandidat-Verbindungen im Connection-Pool findet, dann ruft er die Methode *ManagedConnectionFactory.matchManagedConnection()* im Resource-Adapter auf, wie die Abbildung 6.2 zeigt. Falls keine passende Verbindung gefunden wurde, ruft der Applikationsserver die Methode *ManagedConnectionFactory.matchManagedConnection()* auf, wie die Abbildung 6.1 darstellt.

Die Methode *ManagedConnectionFactory.matchManagedConnection()* wird von dem Resource-Adapter implementiert. Der folgende Beispielcode zeigt, wie der Resource-Adapter diese Methode implementiert. Der Applikationsserver ruft diese Methode auf, übergibt die Kandidat-Verbindungen als Parameter *connectionSet*, und die Sicherheitsinformationen, die von der Applikationskomponente gegeben wurde, als Parameter *info* (im „component

managed Sign-on“). Der Parameter *subject* enthält die Sicherheitsinformationen, die vom Applikationsserver übergeben wurden(im „container managed Sign-on“).

```
public ManagedConnection matchManagedConnections(
                            java.util.Set connectionSet,
                            Subject subject,
                            ConnectionRequestInfo info)
    throws ResourceException
{
  PasswordCredential newpc = new PasswordCredential (
    (( WorkflowConnectionRequestInfo ) info ).getUser(),
    (((WorkflowConnectionRequestInfo)info).getPassword()).toCharArray());
  Iterator it = connectionSet.iterator();
  while ( it.hasNext() )
  {
    Object obj = it.next();
    if (obj instanceof WorkflowManagedConnection)
    {
      WorkflowManagedConnection mc = (WorkflowManagedConnection) obj;
      //get the ManagedConnectionFactory instance, which associates with
      //the candidate ManagedConnection instance
      ManagedConnectionFactory mcf = mc.getManagedConnectionFactory();

      //check if the candidate ManagedConnection instance
      //matches the current request
      if ( newpc.equals( mc.getPasswordCredential() ) && mcf.equals(this))
      {
        return mc;
      }
    }
  }
  return null;
}
```

Eine *ManagedConnection*-Instanz ist immer mit einer *PasswordCredential*-Instanz und einer *ManagedConnectionFactory*-Instanz verbunden. Hier werden die beiden Instanzen vom Resource-Adapter verwendet, um eine geeignete Verbindung für die Verbindungserzeugungsanfrage zu finden.

Eine *PasswordCredential*-Instanz ist ein Behälter für Benutzername und Passwort. Hier wird zuerst der Benutzername und das Passwort vom Parameter *info* ausgelesen, und dann in eine *PasswordCredential*-Instanz gepackt.

Der Resource-Adapter vergleicht die *PasswordCredential*-Instanz und die *ManagedConnectionFactory*-Instanz, die mit einer Kandidat-Verbindung verbunden sind, mit der neu erzeugten *PasswordCredential*-Instanz und der aktuellen *ManagedConnectionFactory*-Instanz. Wenn sie identisch sind, dann liefert der Resource-

Adapter diese *ManagedConnection*-Instanz zurück. Wenn es keine passende *ManagedConnection*-Instanz gibt, dann liefert der Resource-Adapter *null* zurück. Falls der Applikationsserver *null* bekommt, dann ruft er weiter die Methode *ManagedConnectionFactory.createManagedConnection()* auf, um eine neue *ManagedConnection*-Instanz zu erzeugen.

Die Methode *createManagedConnection()* liefert eine Instanz von *ManagedConnection* zurück. Die *ManagedConnectionFactory*-Instanz erzeugt zuerst eine physikalische Verbindung zu einem MQWF-Server. Nachdem diese Verbindung erfolgreich erzeugt wurde, wird eine neue *ManagedConnection*-Instanz, die mit dieser physikalischen Verbindung eng verbunden ist, von der *ManagedConnectionFactory*-Instanz erzeugt.

Um eine Verbindung zu einem MQWF-Server zu erstellen, muss sich die *ManagedConnectionFactory*-Instanz zuerst auf dem MQWF Administration-Server einloggen, mit dem die Applikationskomponente kommunizieren möchte. Die *ManagedConnectionFactory*-Instanz benutzt die Sicherheitsinformationen(Benutzername, Passwort) für das Einloggen auf MQWF Administration-Server. Beim Einloggen wird der Benutzer von dem MQWF Administrations-Server authentifiziert. Nach erfolgreichem Einloggen wird eine Session (ExecutionService) zwischen dem Benutzer und dem MQWF Execution-Server erstellt. Der MQWF Execution-Server wird von dem Systemnamen und dem System-Group-Namen identifiziert.

```
Agent agent = new Agent();
agent.setLocator(locator policy);
agent.setName("agentname");
ExecutionService service = agent.locate("system group", "system");
service.logon(userid, passwd);
service.programExecutionAgentStartUp();
```

Ein Agent repräsentiert eine MQWF-Domain. Eine MQWF-Domain ist eine Gruppe von MQWF-Systemen. Ein MQWF-System enthält Execution-Server, Scheduling-Server, Cleanup-Server usw..

Ein Client benutzt das MQSeries-Workflow-Java-API, um mit dem MQWF-Server zu kommunizieren. Das MQSeries-Workflow-Java-API besteht aus den Java-API-Klassen und einem Java-Agent. Die Java-API-Klassen basieren auf dem C++-API. Das Java-API adaptiert das C++-API über das Java-Native-Interface (JNI) in eine Form, die für die Java-Umgebung zugänglich ist. Eine Locator-Policy ist das Kommunikationsprotokoll, das vom Java-Agent benutzt wird. Um mit einem Java-Agent zu kommunizieren, muss ein Client zuerst den Java-Agent lokalisieren. Der Client benutzt dieselbe Locator-Policy, die der Java-Agent verwendet, um den Java-Agent zu lokalisieren. Die Methode *Agent.setLocator()* wird aufgerufen, um den Typ der Locator-Policy einzugeben, zum Beispiel LOC_LOCATOR für das Local-Binding oder JNDI_LOCATOR für die JNDI-Locator-Policy. Beim Local-Binding verwendet ein Client die C++-API direkt über das Java-Native-Interface. Im JNDI-Locator verwendet der Client das C++-API nicht direkt. Er leitet dem Java-Agent die Methode aufrufe weiter. Mit der Methode *Agent.setName()* wird der Name des Agents eingegeben, mit dem der Client

kommunizieren möchte. Die Locator-Policy muss immer vor dem Agent-Name eingegeben werden.

Die Methode *Agent.locate()* wird aufgerufen, um ein ExecutionService-Objekt im gegebenen System-Group und System zu lokalisieren.

Um die Aktivitätsprogramme einer Prozess-Instanz auszuführen, muss zuerst der Program-Execution-Agent(PEA) gestartet werden. Ein PEA ist für die Ausführungen der Aktivitätsprogramme zuständig. Vom Execution-Service kann der PEA für den eingeloggten Benutzer gestartet werden. Der PEA laüft auf der Maschine, wo das Aufrufprogramm läuft.

Nachdem der Applikationsserver eine *ManagedConnection*-Instanz erhalten hat, wird der Applikationsserver zuerst einen *ConnectionEventListener* bei dieser Instanz registrieren, und anschließend die Methode *ManagedConnection.getConnection()* aufrufen. Um den ConnectionEventListener bei der ManagedConnection-Instanz zu registrieren, ruft der Applikationsserver die Methode *ManagedConnection.addConnectionEventListener()* auf. Wenn eine *ManagedConnection*-Instanz erzeugt wird, wird eine leere Liste für die Listener in Form einer Vector-Instanz erzeugt. Wenn die Methode *addConnectionEventListener()* aufgerufen wird, wird ein Listener in die Liste hinzugefügt.

Die Methode *ManagedConnection.getConnection()* liefert eine *Connection*–Instanz zurück. Diese *Connection*–Instanz wird zu der Applikationskomponente weitergeleitet.

6.2 Abbau einer Verbindung

Eine Applikationskomponente ruft die Methode *Connection.close()* im Resource-Adapter auf, um die Verbindung zu dem MQWF-Server abzubauen. Die Verbindung auf Applikationsebene wird von dem Resource-Adapter gelöscht. Die physikalische Verbindung zu dem MQWF-Server wird im Managed-Applikationsszenario nicht gleichzeitig abgebaut. Der Applikationsserver ruft die Methode *ManagedConnection.destroy()* auf, um die physikalische Verbindung abzubauen. Wie Abbildung 6.3 darstellt.

Die *Connection*-Instanz leitet diese Anfrage innerhalb des Resource-Adapters zu der *ManagedConnection*-Instanz, die mit dieser *Connection*-Instanz verbunden ist, weiter. Die *ManagedConnection*-Instanz schickt dem Applikationsserver eine Nachricht, dass diese Verbindung auf Applikationsebene gelöscht wird. Hier benutzt die *ManagedConnection*-Instanz alle Listener in der Liste, um dem Applikationsserver über das Ereignis zu benachrichtigen.

```
ConnectionEvent ce = new ConnectionEvent(ManagedConnection,
                                    ConnectionEvent.CONNECTION_CLOSED);
ce.setConnectionHandle(connection);
ConnectionEventListener.connectionClosed(ce);
```

Hier im Beispielcode wird zuerst eine *ConnectionEvent*-Instanz mit dem Ereignis CONNECTION_CLOSED erzeugt. Dann wird diese Instanz mit der *Connection*-Instanz assoziiert, die von der Applikationskomponente beendet wurde. Diese *ConnectionEvent*-

Instanz wird als Parameter der Methode *ConnectionEventListener.connectionClosed()* verwendet, um dem Applikationsserver das Ereignis mitzuteilen.

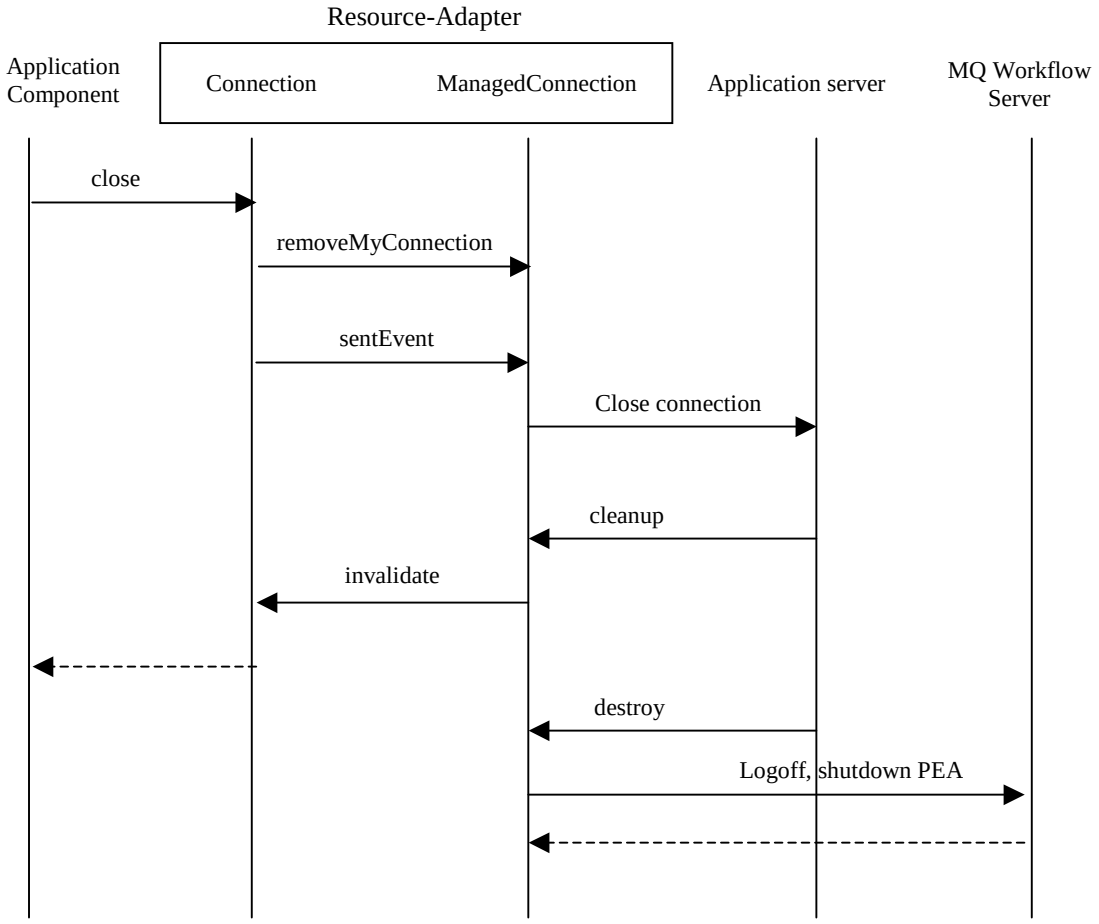


Abbildung 6.3: Verbindungsbeenden im Managed-Applikationsszenario

Wie die Abbildung 6.3 zeigt, ruft der Applikationsserver die Methode *ManagedConnection.cleanup()* im Resource-Adapter auf, um die Verbindung auf Applikationsebene zu beenden. Beim "cleanup" löst die *ManagedConnection*-Instanz die Assoziation mit allen *Connection*-Instanzen, die mit ihr verbunden sind. Hier ruft die *ManagedConnection*-Instanz die Methode *Connection.invalidate()* auf. Die Methode *invalidate()* wird folgendermaßen implementiert: `managedConnection = null;`

Der Applikationsserver ruft die Methode *ManagedConnection.destroy()* im Resource-Adapter auf, um eine physikalische Verbindung abzubauen. Hier wird die Verbindung zu dem MQWF-Server beendet. Der PEA wird zuerst ausgeschaltet und der Benutzer wird auf dem MQWF-Server ausgeloggt.

```
ExecutionService.programExecutionAgentShutDown();
ExecutionService.logoff();
```

Eine ExecutionService-Instanz repräsentiert die Session zwischen dem Benutzer und dem MQWF-Execution-Server. Bevor der Benutzer auf dem Server sich ausloggt, wird der

Program-Execution-Agent zuerst ausgeschaltet, da der Program-Execution-Agent immer mit einem eingeloggten Benutzer assoziiert ist. Mit der Methode *ExecutionService.logoff()* wird die Session zwischen dem Benutzer und dem MQWF-Execution-Server beendet. Wenn das Ausloggen erfolgreich war, wird keine Client/Server-Kommunikation mehr erlaubt, die auf dieser ExecutionService-Instanz basiert.

6.3 Kommunizieren mit MQWF

Der Resource-Adapter für MQWF implementiert auch die standardisierten Common-Client-Interfaces, bietet den Applikationskomponenten die Möglichkeiten, auf den MQWF-Server zuzugreifen und eine Prozess-Instanz auf dem MQWF-Server zu erzeugen und auszuführen.

In MQWF Buildtime kann ein Workflow-Prozess definiert werden. Auch das Personal und die Programme, die mit den Aktivitäten verbunden sind, können in Buildtime definiert werden. Die Workflow-Prozess-Definition wird dann in die Runtime-Komponente exportiert, und ein Template für diese Wokflow-Prozess-Definition wird erzeugt.

MQWF bietet verschiedene APIs, mit denen ein Client mit dem MQWF-Server kommunizieren kann. Hier, in diesem Resource-Adapter für MQWF, wird die Java-API verwendet. Um mit dem MQWF-Server zu kommunizieren, muss sich ein Client zuerst auf dem Server einloggen und von dem Server authentifiziert werden. Der Client muss die notwendigen Input-Daten eingeben, um eine Prozess-Instanz, die von einem Template erzeugt wurde, auszuführen. Nach der Ausführung liefert eine Prozess-Instanz einen Output-Container zurück. Der Resource-Adapter bildet diesen Output-Container auf eine *IndexedRecord*-Instanz ab und liefert der Applikationskomponente diese *IndexedRecord*-Instanz zurück.

Die Applikationskomponenten können den Resource-Adapter verwenden, um die folgenden Schritten zu realisieren:

- Sicherheitsinformationen(z.B. Benutzername, Passwort) zu übertragen, die für das Einloggen auf dem MQWF-Server notwendig sind;
- Objekten(z.B. die Prozess-Templates, die Prozess-Instanzen usw..) Zugriffe auf den MQWF-Server zu ermöglichen,
- die notwendigen Informationen (z.B. Templatenamen) einzugeben, um eine Prozess-Instanz von entsprechenden Templates zu erzeugen;
- die Input-Daten einzugeben, um eine Prozess-Instanz auszuführen;
- die Output-Daten auszulesen, die nach Ausführung der Prozess-Instanz zurückgeliefert wurden;
- die Verbindung zu beenden;

WorkflowConnectionSpec

Die Klasse *WorkflowConnectionSpec* implementiert das Interface *ConnectionSpec*. Diese Klasse bietet der Applikationskomponente die Möglichkeit, die Sicherheitsinformationen in eine *WorkflowConnectionSpec*-Instanz zupacken.

```
public class WorkflowConnectionSpec
```

```
   implements javax.resource.cci.ConnectionSpec
{
  private String user;
  private String password;
  public WorkflowConnectionSpec() {}
  public WorkflowConnectionSpec(String user, String password)
  {
    this.user = user;
    this.password = password;
  }
  public String getPassword()
  {
    return password;
  }
  public String getUser()
  {
    return user;
  }
}
```

Die Applikationskomponente kann eine neue *WorkflowConnectionSpec*-Instanz mit dem Benutzernamen und dem Passwort erzeugen: *new WorkflowConnectionSpec(user, pssswd).* Diese Informationen werden später von dem Resource-Adapter ausgepackt, und für das Einloggen auf dem MQWF-Server verwendet.

WorkflowInteractionSpec

Die Applikationskomponente benutzt die Klasse *WorkflowInteractionSpec*, um den Templatenamen einzugeben, auf dessen Template die Applikations-Komponente zugreifen möchte.

```
public class WorkflowInteractionSpec
   implements InteractionSpec, java.io.Serializable
{
  String templateName;

  public WorkflowInteractionSpec() {}
  public String getTemplateName()
  {
    return templateName;
  }
  public void setTemplateName(String tempName)
  {
    templateName = tempName;
  }
}
```

In dieser Klasse werden nur getter- und setter-Methoden bereitgestellt, damit die Applikationskomponente den Templatename einlesen und der Resource-Adapter den Templatename auslesen kann. Mit diesem Templatenamen kann ein Resource-Adapter das entsprechende Template auf dem MQWF-Server finden.

WorkflowIndexedRecord

Die Klasse *WorkflowIndexedRecord* implementiert das Interface *IndexedRecord*. Diese Klasse ermöglicht der Applikationskomponente, die Input-Daten für eine Prozess-Instanz einzugeben und die Rückgabewerte von der Prozess-Instanz auszulesen.

WorkflowDataType

Die Klasse *WorkflowDataType* definiert einen Datentyp, das einem Container-Blatt in MQWF entspricht. Die Applikationskomponente kapselt die Input-Daten zuerst in den *WorkflowDataType*-Instanzen. Dann werden diese Instanzen als Objekte in einer *WorkflowIndexedRecord*-Instanz hinzugefügt.

In MQWF repräsentiert ein Container die Input- oder Output-Daten für das Prozess-Template, den Prozess-Instanzen, Work-Items oder der Aktivitäts-Implementierung. Ein Container wird durch eine Datenstruktur definiert, wie es in der Abbildung 6.4 gezeigt wird.

Eine Datenstruktur hat mehrere Daten-Members. Ein Daten-Member hat einen Namen und einen Datentyp. Der Datentyp ist entweder eines der Basisdatentypen STRING, LONG, BINARY, FLOAT oder eine andere Datenstruktur. Hier darf ein Daten-Member rekursiv definiert werden. Ein Daten-Member hat ein *fully-qualified-name*, der den hierarchischen Daten-Member-Namen, zum Beispiel Addr.street, beschreibt.

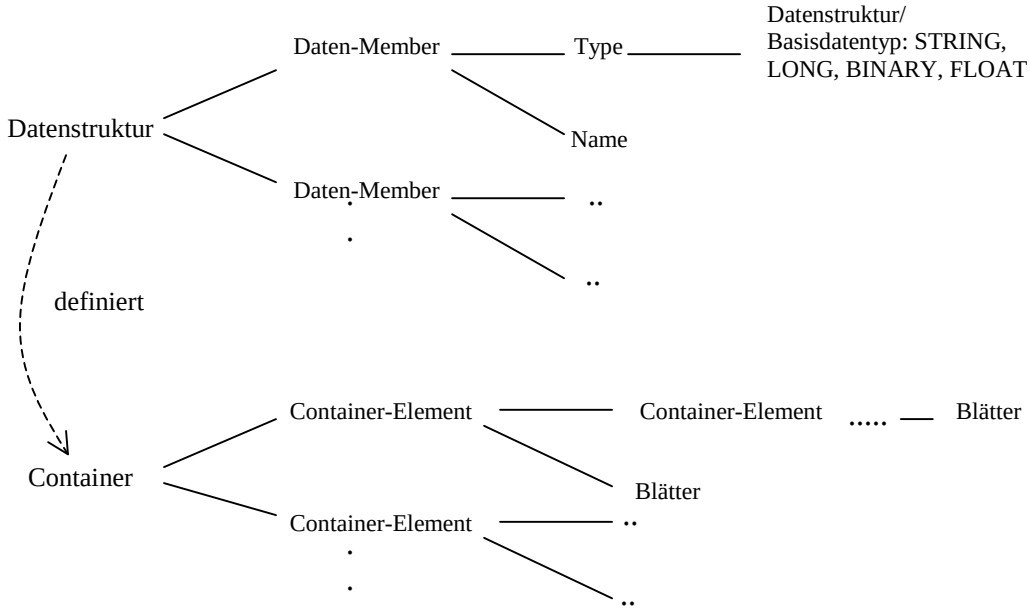


Abbildung 6.4: Abbildung einer Datenstruktur auf einen Container

Eine Datenstruktur stellt ein Container dar, während die Daten-Member einer Datenstruktur Container-Elemente darstellen. Falls der Datentyp eines Container-Elements eine

Datenstruktur ist, dann enthält dieses Container-Element wieder ein Container-Element. Container-Elemente, die Basisdatentype sind, heißen Blätter eines Container. Blätter von Container dürfen ein Wert enthalten.

Hier ist ein Beispiel für eine Datenstruktur. Die Datenstruktur PERSON beschreibt einen Input- oder Output-Container. Die Struktur ist so definiert:

```
Name      STRING
Addr      ADDRESS
            Street    STRING
            PLZ       LONG
```

PERSON hat zwei Daten-Member, nämlich Name und Addr. Der Daten-Member *Name*, welcher vom Basisdatentyp STRING ist, enthält keine weitere Hierarchieebenen. Der *Addr*-Daten-Member ist dagegen vom Datentyp ADDRESS, der wiederum eine Datenstruktur mit den zwei Daten-Members Street und PLZ repräsentiert. Street ist hier vom Basistdatentyp STRING, und PLZ vom Basisdatentyp LONG.

Ein Input- oder Output-Container, der von der Datenstruktur PERSON definiert ist, hat die Blätter mit dem *fully-qualified-name* wie folgt:

```
Name         STRING
Addr.Street  STRING
Addr.PLZ     LONG
```

Die Klasse *WorkflowDataType* simuliert die Struktur eines Container-Blattes. Die Attribute *name*, *type* und *value* im *WorkflowDataType* entsprechen dem *fully-qualified-name*, dem Datentyp und dem Wert des Container-Blattes. Die Applikationskomponente kann diese Klasse benutzen, um die Input-Daten für einen Input-Container in den Objekten einzupacken. Die Klasse bietet auch Methoden, mit denen die Applikationskomponente die Daten auslesen kann.

```
public class WorkflowDataType
{
  private String name;
  private String type;
  private Object value;
  public WorkflowDataType (String name, String type, Object value)
  {
    this.name = name;
    this.type = type;
    this.value = value;
  }
  public String getName ()
  {
    return name;
  }
  public String getType ()
```

```
  {
    return type;
  }
  public String getString()
  {
   return (String) value.toString();
  }
  public int getLong()
  {
   return ((Integer)value).intValue();
  }
  public double getDouble()
  {
   return ((Double)value).doubleValue();
  }
  public byte[] getBuffer()
  {
    return (byte[]) value;
  }
}
```

Mapping

Wenn eine Prozess-Instanz auf dem MQWF-Server erfolgreich ausgeführt wurde, wird ein OutContainer zurückgeliefert. Der OutContainer wird dann von der Klasse Mapping auf eine *IndexedRecord*-Instanz abgebildet.

Zuerst werden alle Container-Blätter von dem OutContainer ausgelesen und in ein Array gelegt.

```
ContainerElement[] leaves = outContainer.leaves( );
```

Von jedem Element des Arrays werden der *fully-qualified-name*, der DatenTyp und der entsprechende Wert ausgelesen, und wieder in ein *WorkflowDataType*-Objekt eingepackt. Zum Beispiel:

```
String fullName = leaves[ i ].fullName( );
String type = leaves[ i ].type( );
if ( type.equals( "STRING" ) )
{
  Object obj = new Object ( (String) leaves[ i ].getString( ) );
}
WorkflowDataType wfdata = new WorkflowDataType( fullName, type, obj);
```

Jedes *WorkflowDataType*-Objekt wird schließlich einer *IndexedRecord*-Instanz hinzugefügt, die am Ende zurückgeliefert wird.

```
indexedRecord.add(wfdata);
return indexedRecord;
```

WorkflowInteraction

Die Klasse *WorkflowInteraction* implementiert das Interface *javax.resource.cci.Interaction*. Diese Klasse bietet den Applikationskomponenten eine wichtige Methode. Mit dieser Methode kann die Applikationskomponente auf Templates im MQWF-Server zugreifen, Input-Daten übertragen und letztendlich Output-Daten holen.

Basierend auf der Benutzersession(ExecutionService) zwischen dem eingeloggten Benutzer und dem MQWF-Server werden die folgenden Schritte in der Methode *Interaction.execute()* durchgeführt:

- Auspacken des Templatenamen aus der *InteractionSpec*-Instanz;
- Finden des Prozess-Templates über den Templatenamen;

```
ProcessTemplate[] templates = ExecutionService.queryProcessTemplates(
                            "NAME='" + templateName + "'", null, null)
```

- Initialisieren des Input-Containers des Prozess-Templates mit den *WorkflowDataType*-Objekten der *IndexedRecord*-Instanz;

```
ReadWriteContainer inCnr = ProzessTemplate.initialInContainer();
```

  Zuerst wird der Input-Container, der mit diesem Prozess-Template assoziiert ist, von dem MQWF Execution-Server geholt.

  Dann werden die Input-Daten, die in einem *WorkflowDataType*-Objekt eingepackt wurden, in den Input-Container geschrieben. Zum Beispiel werden so die Daten vom Datentyp STRING in einen Input-Container geschrieben.

```
inCnr.setString( WorkflowDataType.name,
                 WorkflowDataType.getString( ));
```

- Erzeugen einer Prozess-Instanz, Ausführen der Prozess-Instanz, und schließlich Erhalten des Output-Containers.

```
ReadOnlyContainerHolder outCnrHol = new ReadOnlyContainerHolder();
ProcessInstance pi = template.executeProcessInstance2(
                              inCnr, outCnrHol, "", "", "", false);
outCnr = outCnrHol.value;
```

- Abbilden des Output-Containers auf eine *IndexedRecord*-Instanz mittels der Klasse Mapping;
- Liefern der *IndexedRecord*-Instanz als Rückgabewert.

6.4 Security Management

Im Component-Managed-Sign-On müssen die Applikationskomponenten dem Resource-Adapter die Sicherheitsinformationen (Benutzername und Passwort) übergeben, um sich auf dem MQWF-Server unter diesem Benutzername einzuloggen, und schließlich eine Verbindung zu dem MQWF-Server herzuerstellen. Das wurde in Abschnitt 6.1 genau beschrieben.

In einer Prozess-Definition muss jeder Prozess und jede Aktivität mit Personen oder Rollen assoziiert sein. Diese Personen oder Rollen müssen zuvor im Build-Time mit UserID und Passwort definiert werden. Im Run-Time wird ein Benutzer von dem MQWF-Server authentifiziert und autorisiert, um zu überprüfen, ob er das Recht hat, auf eine Prozess oder eine Aktivität zuzugreifen.

Eine Applikationskomponente verwendet das Interface *ConnectionReqestInfo*, um dem Resource-Adapter den Benutzername und Passwort zu übergeben. Dieser Benutzer wird nicht vom WAS, sondern nur vom MQWF-Server beim Einloggen authentifiziert. Nachdem sich der Benutzer auf dem MQWF-Server erfolgreich eingeloggt hat, darf die Applikation auf die dem Benutzer autorisierten Prozesse oder Aktivitäten zugreifen.

6.5 Transaction Management

Dieser Resource-Adapter unterstützt keine Transaktionen, weil MQWF keine Transaktion unterstützt. Die folgenden Methoden müssen im Interface *Connection* und *ManagedConnection* die entsprechende *NotSupportedException*-Ausnahme werfen:

```
public class WorkflowConnection
       implements javax.resource.cci.Connection
{
  public javax.resource.cci.LocalTransaction getLocalTransaction()
     throws   ResourceException
  {
    throw new NotSupportedException("Local Transaction not supported!");
  }
}


public class WorkflowManagedConnection
       implements javax.resource.spi.ManagedConnection
{
  public javax.resource.spi.LocalTransaction getLocalTransaction()
     throws javax.resource.ResourceException
  {
    throw new NotSupportedException( "Local transaction not supported." );
  }
```

```
public javax.transaction.xa.XAResource getXAResource()
   throws ResourceException
  {
    throw new NotSupportedException( "XA transaction not supported." );
  }
}
```

7 Non-managed-Applikationsszenario

Die Connector-Architektur definiert den System-Kontrakt zwischen dem Resource-Adapter und dem Applikationsserver. Sie unterstützt auch die Non-Managed-Applikationsszenario. Das heißt, die Applikationen, z.B Java-Applikationen und Applets, benutzen direkt den Resource-Adapter, um eine Verbindung zu einem EIS zu erzeugen, mit dem EIS zu kommunizieren, und schließlich die Verbindung zu beenden. Die Applikationen sorgen auch selbst für die Sicherheit und für die Transaktionen. Im Managed-Applikationsszenario bietet der Applikationsserver Dienste für Sicherheit, Transaktionen und Connection-Pooling.

7.1 Die Struktur des Non-managed-Applikationsszenarios

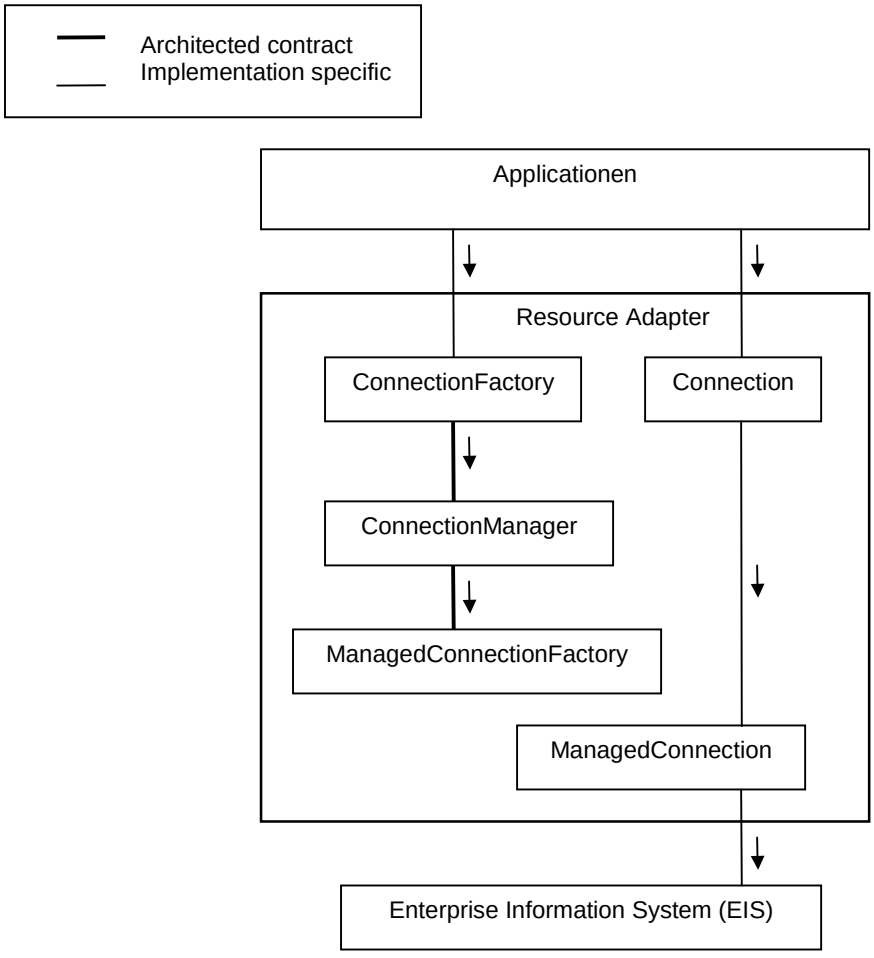


Abbildung 7.1[CONSPE10]: Die Architektur des Non-Managed-Applikationsszenarios

Die Abbildung 7.1[CONSPE10] zeigt die Struktur eines Non-Managed-Applikationsszenarios. Wie im Managed-Applikationsszenario bietet der Resource-Adapter den Applikationen im Non-Managed-Applikationsszenerio auch die Interfaces *ConnectionFactory* und *Connection* an. Aber im Non-Managed-Applikationsszenario muss ein Resource-Adapter das Interface *ConnectionManager* noch zusätzlich implementieren, während im Managed-

Applikationsszenario der Applikationsserver dieses Interface implementiert. Der ConnectionManager ist zuständig für die Verbindungslokalisierung.

Falls ein Resource-Adapter intern kein Connection-Pooling implementiert, werden die *ManagedConnection*-Instanzen nach Erzeugen nicht in einen Connection-Pool hinzugefügt.

7.2 Verwendung des Resource-Adapters

7.2.1 Erzeugung einer *ConnectionFactory*-Instanz

Im Managed-Applikationsszenario werden die *ManagedConnectionFactory*- und *ConnectionFactory*-Instanz beim Deployment vom Deploymentstool (z.B. WebSphere Administration console) erzeugt, während im Non-Managed-Applikationsszenario die beide Instanzen von den Applikationen erzeugt werden. Zuerst verwenden die Applikationen die Resource-Adapter-Library, um eine *ManagedConnectionFactory*-Instanz zu erzeugen. Dann benutzen die Applikationen diese Instanz, um eine *ConnectionFactory*-Instanz zu erstellen. Der folgende Beispielcode zeigt, wie eine Applikation diese beide Instanzen erzeugt:

```
//erzeugt eine neue Instanz von ManagedConnectionFactory-
//implementierungsklasse WorkflowManagedConnectionFactory
WorkflowManagedConnectionFactory mcf =
     new WorkflowManagedConnectionFactory();

//set die Properties auf der ManagedConnectionFactory-Instanz, zum Beispiel
mcf.setEisName(…);

//erzeugt eine ConnectionFactory-Instanz
ConnectionFactory cf = (ConnectionFactory) mcf.createConnectionFactory();
```

Wie es in dem Beispielcode gezeigt wird, wird die Methode *ManagedConnectionFactory.createConnectionFactory()* aufgerufen, um eine *ConnectionFactory*-Instanz zu erzeugen. Diese Methode verlangt keine Parameter. In diesem Fall bietet die *ManagedConnectionFactory*-Instanz eine default *ConnectionManager*-Instanz an, die von dem Resource-Adapter oder den Applikationsentwicklern implementiert wurde, um eine *ConnectionFactory*-Instanz zu erzeugen. Im Managed-Fall ruft der Applikationsserver diese Methode auf, und übergibt eine *ConnectionManager*-Instanz als Parameter.

7.2.2 Registrieren einer *ConnectionFactory*-Instanz im JNDI-Namespace

Bevor eine Applikation ein Objekt im JNDI-Namespace finden kann, muss das Objekt im JNDI-Namespace zuerst registriert werden. Im Managed-Applikationsszenario wird eine *ConnectionFactory*-Instanz nach Erzeugung von dem Deploymentstool in JNDI-Namespace gespeichert. Im Non-Managed-Applikationsszenario muss die Registrierung der *ConnectionFactory*-Instanz von der Applikation oder einer Hilfeklasse implementiert werden.

In der Connector-Architektur wurden zwei Mechanismen definiert, um eine *ConnectionFactory*-Instanz im JNDI-Namespace zu registrieren. Sie sind JNDI-Reference- und JNDI-Serializable-Mechanismen[CONSPE10, Kapitel 10, Abschnitt 10.5]. Um die

beiden Mechanismen zu unterstützen, muss eine *ConnectionFactory*-Klasse das Interface *java.io.Serializable* und *javax.resource.Referenceable* implementieren.

Um den Reference-Mechanismus im Non-Managed-Applikationsszenario zu unterstützen, muss ein Resource-Adapter oder eine Hilfeklasse das Interface *javax.naming.spi.ObjectFactory* implementieren.

7.2.3 Finden einer *ConnectionFactory*-Instanz

Im Managed-Applikationsszenario benutzt die Applikationskomponente die *lookup()*-Methode um eine *ConnectionFactory*-Instanz im Namespace zu finden. Um die Konsistenz des Applikationsprogrammierungsmodells unter Managed- und Non-Managed-Applikationsszenarien beizubehalten, sollen die Applikationen im Non-Managed-Szenario auch das JNDI-Namespace benutzen, um eine *ConnectionFactory*-Instanz zu finden[CONSPE10].

```
Properties env = new Properties();
//using webSphere Server naming service
env.put( Context.INITIAL_CONTEXT_FACTORY,
         "com.ibm.websphere.naming.WsnInitialContextFactory" );
env.put( Context.PROVIDER_URL, "iiop://localhost:900" );

javax.naming.InitialContext ctx = new javax.naming.InitialContext(env);
ConnectionFactory cf =
       ( ConnectionFactory )ctx.lookup( "conFactory" );
```

Hier im Beispielcode wird der IBM WebSphere Applikationsserver als Service-Provider verwendet.

Ein Objekt muss zuerst im Namespace registriert werden, dann kann eine Applikation das Objekt wieder im Namespace finden.

7.2.4 Registrierungsmechanismen

Die Connector-Architektur definiert das *ConnectionFactory*-Interface so, dass es den Serializable- und den Referenceable-Mechanismus unterstützt.

Serializable

Wenn eine Java-Klasse das Interface *java.io.Serializable* implementiert, dann ist das Java-Object serialisierbar. Das heißt, die Informationen, die das Objekt identifizieren, können in einen Byte-Strom umgewandelt und im JNDI-Namespace gespeichert werden, damit eine Kopie des Objektes später wieder von dem Byte-Strom erstellt werden kann.

Der folgende Beispielcode zeigt, wie eine *Serializable ConnectionFactory*-Instanz im JNDI-Namespace registriert wird:

```
javax.naming.InitialContext ctx = new javax.naming.InitialContext(env);
```

```
ConnectionFactory cf = (ConnectionFactory) mcf.createConnectionFactory();
//bind cf mit JNDI Name "conFactory"
ctx.bind("conFactory"", cf);
```

Mit der Methode *lookup()* kann dieses Objekt später wieder aus dem JNDI-Namespace ausgelesen wird.

```
ConnectionFactory cf = (ConnectionFactory) ctx.lookup( "conFactory" );
```

Referenceable

Nicht nur eine Kopie eines Objektes kann im JNDI-Namespace gespeichert werden, sondern auch eine Referenz eines Objektes. Eine Referenz besteht aus der Adresse und den Klasseninformationen über ein *referenceable* Objekt. Die Adresse enthält die Informationen, um ein Objekt wieder aufzubauen. Hier wird *BinaryRefAddr* für die Adresse verwendet, und alternativ *StringRefAddr*. Eine *BinaryRefAddr* enthält den Adress-Typ und das Byte-Array des Objektes als Inhalt. Eine *StringRefAddr* enthält den Adress-Typ und einen String als Inhalt. Eine Referenz enthält auch den Klassennamen einer *ObjectFactory*-Implementierung. Die *ObjectFactory*-Instanz ist für die Erzeugung eines Objektes zuständig. Im Folgenden wird der Beispielcode, um eine Referenz einer *ConnectionFactory*-Instanz zu erzeugen, dargestellt:

```
//Die ConnectionFactory-Instanz wird in einen output-Fluß von den Byten
geschrieben.
ByteArrayOutputStream bOutStream = new ByteArrayOutputStream();
ObjectOutputStream oOutStream = new ObjectOutputStream(bOutStream);
oOutStream.writeObject(cf);
oOutStream.flush();

//Erzeugen einer neuen BinaryRefAddr-Instanz mit Parameter Address-Typ
//und eine byte array für Inhalt.
BinaryRefAddr refAddr = new BinaryRefAddr ( "connectionFactory",
                                            bOutStream.toByteArray() );

//Klassename einer ObjectFactory-Implementation
String objectFactoryClassName =
      new String(ObjectFactory-Implementierungsklassename);

// Erzeugung einer Referenz-Objekt
Reference ref = new Reference( Connection Factory name,
                               refAddr,
                               objectFactoryClassName,
                               null);
```

Wenn eine Klasse das Interface *Referenceable* implementiert, dann hat die Instanz der Klasse eine assoziierende Referenz. Zum Beispiel, die Klasse des *ConnectionFactory*-Interfaces implementiert *javax.resource.Referenceable.* Der Code sieht so aus:

```
public class WorkflowConnectionFactory implements
     javax.resource.Referenceable,
     java.io.Serializable,
     javax.resource.cci.ConnectionFactory
{
  javax.naming.Reference reference;
  public void setReference ( Reference ref )
  {
    reference = ref;
  }
  public Reference getReference() trows NamingException
  {
    return reference;
  }
}
```

Die *Referenceable ConnectionFactory*-Instanz kann im JNDI-Namespace registriert werden:

```
cf.setReference( ref );
ctx.bind( "conFactory", cf);
```

Der Service-Provider hat die Methoden bind() und rebind() so implementiert, dass zuerst die Methode *Referenceable.getReference()* aufgerufen wird, um die Referenz eines Objektes zu bekommen. Dann wird diese Referenz im Namespace gespeichert, nicht das ganze Objekt. Wenn eine Applikation das Objekt mit der *lookup()*-Methode im Namespace aufsucht, wird diese Referenz von der ObjectFactory benutzt, um die Instanz dieses Objektes wiederzugewinnen.

ObjectFactory

Eine *ObjectFactory*-Klasse benutzt Informationen, z.B. eine Referenz, um das Objekt wieder erzeugen zu können. Das *ObjectFactory*-Interface definiert nur eine Methode *getObjectInstance()*. Wenn eine Applikation ein Objekt im Namespace sucht, wird die Methode *ObjectFactory.getObjectInstance()* vom Service-Provider aufgerufen, um ein Objekt zu erhalten. Der folgende Beispielcode zeigt die Implementierung eines *ObjectFactory*-Interfaces:

```
public Object getObjectInstance( Object obj,
                                 Name name,
                                 Context ctx,
                                 Hashtable env ) throws Exception
{
  // The Object parameter is expected to be a Reference
  Reference ref = (Reference)obj;

  //retrieve the first address that has the address type
  //'connectionFactory'
  RefAddr rAddr = ref.get("connectionFactory");
```

```
  //get content from RefAddr
  Object content = rAddr.getContent();

  //construct new ByteAyyarInoutStream
  ByteArrayInputStream bInStream =
              new ByteArrayInputStream((byte[])content);
  ObjectInputStream oInStream = new ObjectInputStream(bInStream);

  // Get the connection factory from the Reference
  Object cf = oInStream.readObject();
  Return cf;
}
```

Zuerst wird gemäss des Adresstyps die Adresse aus einer Referenz gelesen. Diese Adresse enthält ein Byte-Array des Objektes als Inhalt. Der Inhalt wird dann ausgelesen, um ein Objekt wieder herzustellen.

7.2.5 Erzeugung einer Verbindung zu einem EIS

Im Managed-Applikationsszenario wird Connection-Management-Kontrakt zwischen dem Applikationsserver und dem Resource-Adapter definiert. Der Connection-Management-Kontrakt unterstützt Connection-Pooling-Mechanismus. Der Applikationsserver bietet Connection-Pooling-Service, um die Verbindungen zu dem EIS zu verwalten. Im Non-Managed-Applikationsszenario verwaltet der Applikationsclient selbst die Verbindungen zu einem EIS.

Die Abbildung 7.2 zeigt, wie ein Client eine Verbindung zu dem MQWF-Server erzeugt. Das ist ähnlich wie im Managed-Applikationsszenario. Der Applikationsclient ruft zuerst die Methode *ConnectionFactory.getConnection()* im Resource-Adapter auf, wie im Managed-Applikationsszenario. Dann benutzt die *ConnectionFactory*-Instanz die default *ConnectionManager*-Instanz, die vom Resource-Adapter implementiert wurde, um eine Verbindung zu lokalisieren. Falls ein Resource-Adapter intern keinen Connection-Pooling implementiert, dann ruft die *ConnectionManager*-Instanz im Non-Managed-Applikationsszenario direkt die Methode *ManagedConnectionFactory.createManagedConnection()* auf, um eine neue Verbindung zu erzeugen. Während im Managed-Applikationszenario die Methode *ManagedConnectionFactory.matchManagedConnection()* zuerst aufgerufen wird, falls der Applikationsserver die Kandidat-Verbindungen für die Anfrage im Connection-Pool findet.

Wie im Managed-Applikationsszenario erzeugt die *ManagedConnectionFactory*-Instanz eine physikalische Verbindung zu einem EIS. Hier loggt sich der Benutzer auf dem MQWF-Server ein und ein PEA wird für den Benutzer gestartet. Schließlich liefert sie dem ConnectionManager diese *ManagedConnection*-Instanz zurück.

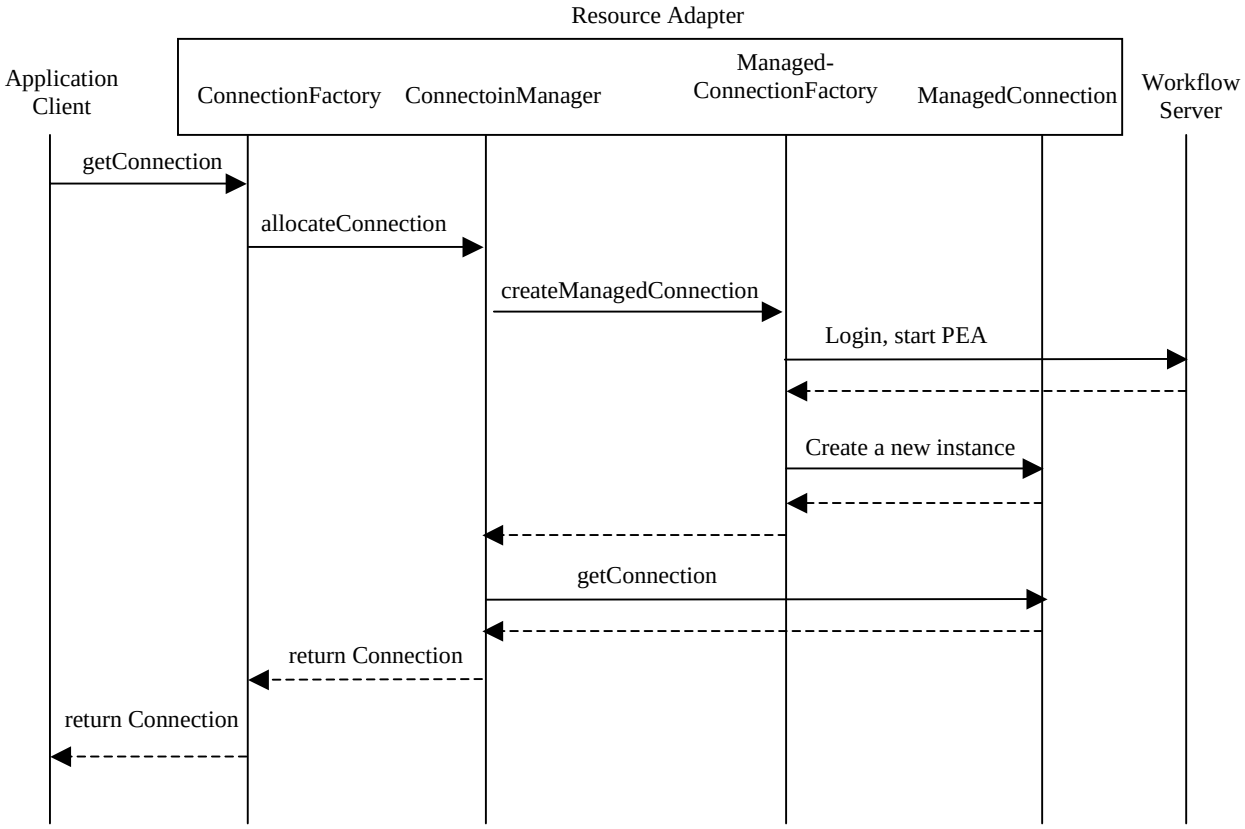


Abbildung 7.2[CONSPE10] : Verbindungserzeugung im Non-Managed-Applicationsszenario

Der ConnectionManager verlangt eine Verbindung auf Applikationsebene von der *ManagedConnection*-Instanz, und leitet die zurückgelieferte Verbindung weiter zu dem Applikationsclient, der eine Verbindung angefordert hat.

Nachdem der Applikationsclient die Verbindung bekommen hat, verwendet er den Resource-Adapter, um auf ein EIS zuzugreifen.

7.2.6 Beenden einer Verbindung

Im Managed-Applikationsszenario beendet die Applikationskomponente nur die Verbindung auf Applikationsebene. Die physikalische Verbindung zu einem EIS wird nicht abgebaut, sondern von dem Applikationsserver in einen Connection-Pool hinzugefügt, und kann wieder verwendet werden. Im Non-Managed-Applikationsszenario wird die physikalische Verbindung von dem Applikationsclient abgebaut, wenn der Resource-Adapter kein Connection-Pooling unterstützt, wie es Abbildung 7.3 zeigt.

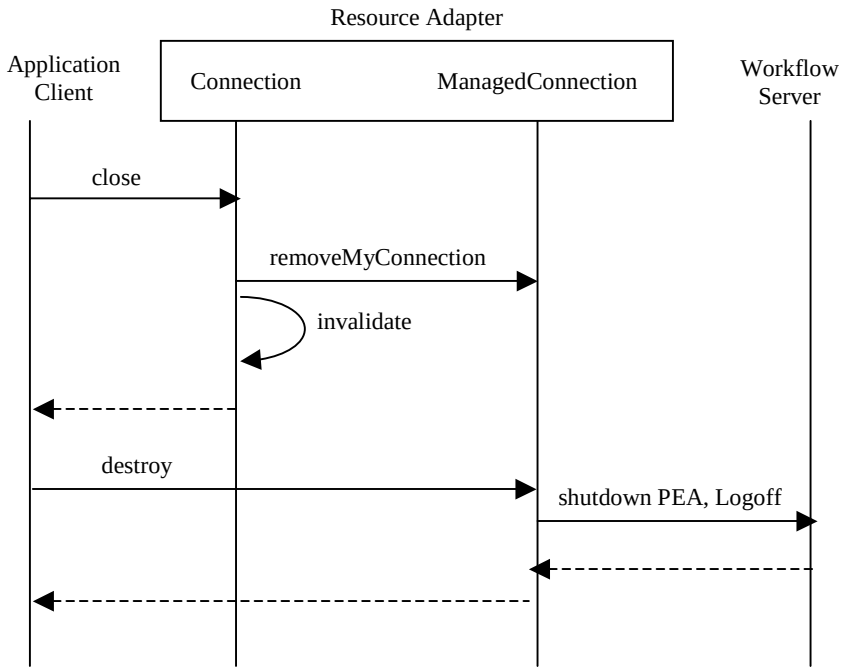


Abbildung 7.3: Verbindungsbeenden im Non-Managed-Applikationsszenario

Im Managed-Applikationsszenario schickt die *ManagedConnection*-Instanz dem Applikationsserver die Nachricht, dass eine Komponente die Verbindung beenden möchte. Dann ruft der Applikationsserver die Methode *cleanup()* auf, um die Verbindung auf Applikationsebene abzubauen, und die physikalische Verbindung in den Connection-Pool hinzuzufügen, wie Abbildung 6.3 darstellt. Im Non-Managed-Applikationsszenario wird die Verbindung auf Applikationsebene in der *Connection*-Instanz beendet. Nachdem der Applikationsclient die Verbindung auf Applikationsebene beendet hat, ruft der Client die Methode *ManagedConnection.destroy()* weiter auf, um die physikalische Verbindung zu dem EIS abzubauen. Hier wird der PEA zuerst beendet und der Client auf dem MQ-Workflow-Server ausgeloggt.

8 Zeitmessungen des Resource-Adapters

Um das Zeitverhalten des Resource-Adapters zu untersuchen, wurden die Zeitmessungen für den Resource-Adapter im Managed- und Non-Managed-Fall durchgeführt. Ein Ziel ist es, die Unterschiede im Zeitverhalten dieser beiden Fällen zu untersuchen, während ein anderes Ziel die Abhängigkeit der Zeitmessung von den Datenvolumina herauszufinden versucht.

8.1 Beispielszenario

Um die Zeitmessungen am Resource-Adapter durchzuführen, werden hier eine Java-Anwendung für das Managed-Applikationsszenario und eine Java-Applikation für das Non-Managed-Applikationsszenario implementiert.

Im Managed-Applikationsszenario wird ein Session-Bean und ein Client des Session-Beans implementiert. Das Session-Bean und der Client werden zusammen in eine Anwendung eingepackt, und auf dem WebSphere-Server installiert. Der Client liest die Daten, die für das Einloggen auf dem MQWF-Server und der Ausführung einer Prozess-Instanz benötigt werden, aus einer Properties-Datei. Der Client ruft eine Methode des Beans remote auf und übergibt dem Bean die Daten. Das Bean benutzt diese Daten, um mit dem Resource-Adapter zu kommunizieren. Nachdem das Bean die vom Resource-Adapter zurückgelieferten Daten bekommen hat, leitet es diese dem Client weiter. Der Client schreibt die Daten wieder in eine Properties-Datei.

Im Non-Managed-Applikationsszenario wird eine Java-Applikation implementiert. Die Applikation verwendet den Resource-Adapter, um sich über den Benutzer auf dem Workflow-Server einzuloggen, die Prozess-Instanz auszuführen, und um sich schließlich wieder auszuloggen. Die dazu benötigten Daten werden von der Applikation aus einer Properties-Datei gelesen. Die vom Resource-Adapter zurückgelieferten Daten werden schließlich auch in einer Properties-Datei gespeichert.

8.2 Messungsumgebung

Im Managed-Fall läuft die Beispielanwendung auf dem WebSphere-Server. Der Resource-Adapter wurde auch auf dem WebSphere-Server installiert. Ein MQWF-Server läuft auf derselben Maschine, auf der auch der WebSphere-Server läuft. Die Anwendung wird mit dem Befehl *launchclient* gestartet.

Im Non-Managed-Fall wird der WebSphere-Server als JNDI-Name-Service-Provider verwendet. Die Java-Applikation benutzt den Resource-Adapter, um auf den MQWF-Server zuzugreifen. Alle Komponenten (MQWF-Server, WebSphere-Server) laufen auf einem Rechner. Die Java-Applikation wird mit dem Befehl *java* ausgeführt.

Ein Messprogramm wurde in Java implementiert, um die Beispielanwendung auszuführen und die Gesamtzeit zu messen, die für die Ausführung der Beispielanwendung gebraucht wird.

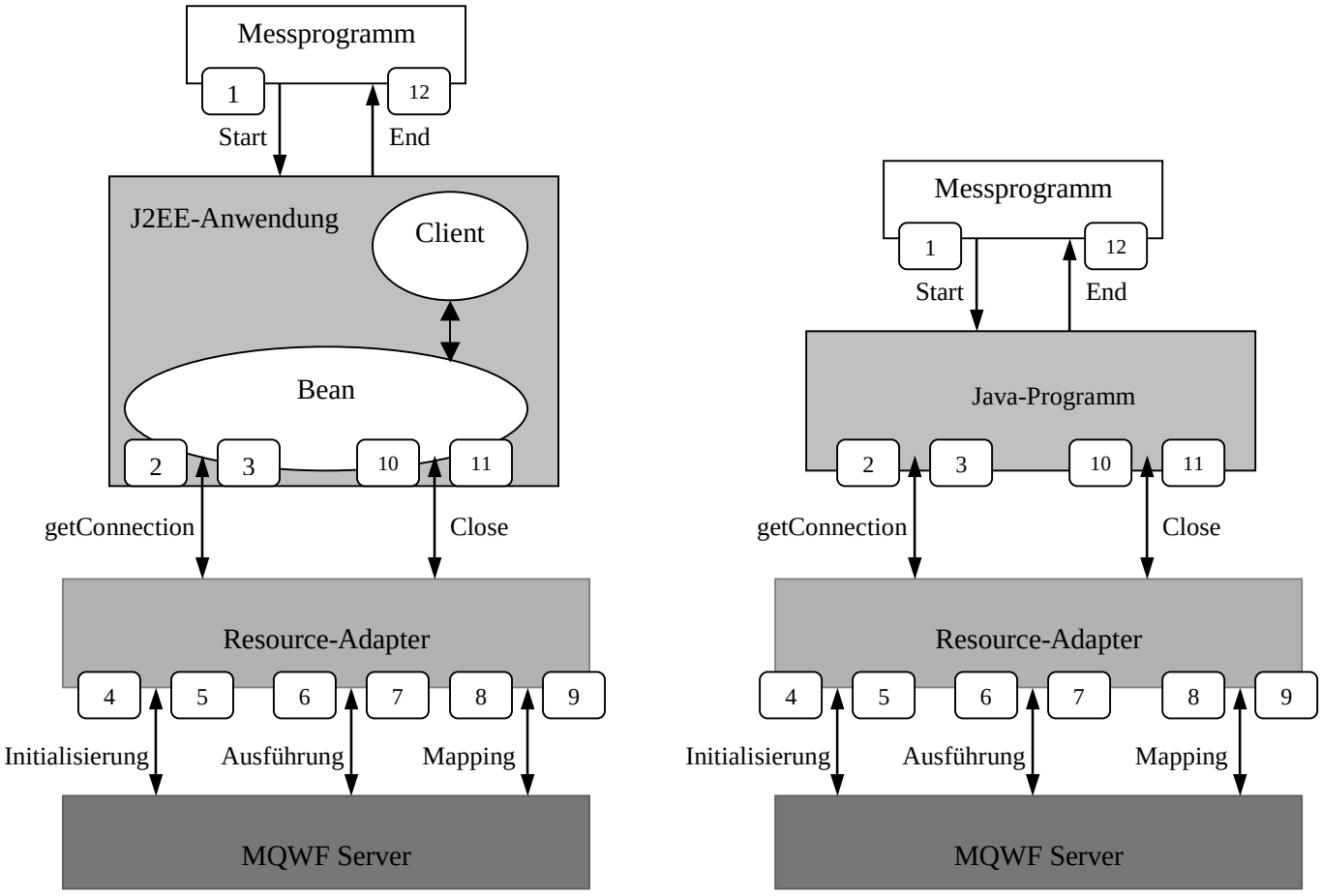
Um ein aussagekräftiges Messergebnis zu bekommen, wurde das Programm mehrmals ausgeführt. In jedem Durchlauf werden die Zeiten zu den bestimmten Messpunkten gemessen. Da es möglich ist, dass zufällige Schwankungen auftreten, die vom aktuellen Zustand des Betriebssystems abhängen, wird der Medianwert der entsprechenden Messdaten genommen, um die Auswirkung starker Schwankungen auf das Messergebnis zu vermeiden.

Als Testrechner wurde ein Intel Pentium III-Rechner mit 850MHZ und 328MB Hauptspeicher verwendet. Als Testplattform diente MS-Windows 2000.

8.3 Messpunkte

Zum Messen wurde *System.currentTimeMillis()* verwendet. Die Messdaten wurden mit *System.out.println()* entweder auf dem Bildschirm(im Non-Managed-Fall) ausgegeben, oder in eine Log-Datei(im Managed-Fall) geschrieben. Für die Managed- und Non-Managed-Applikationsszenarien wurde die Zeit für die folgenden Punkte gemessen, wie es Abbildung 8.1 zeigt:

1. Das Messprogramm startet die Anwendung. Im Managed-Fall wird die Anwendung mit dem Befehl *launchclient* gestartet. Im Non-Managed-Fall wird die Java-Applikation mit *java* gestartet.
2. Das Bean oder die Java-Applikation ruft die Methode *ConnectionFactory.getConnection()* im Resource-Adapter auf, um eine Verbindung zu dem MQWF-Server zu erhalten.
3. Das Bean oder die Java-Applikation hat die Verbindung zu dem MQWF-Server vom Resource-Adapter aufgebaut.
4. Der Resource-Adapter startet die Initialisierung des InContainers von einer Prozess-Instanz.
5. Der Resource-Adapter beendet die Initialisierung des InContainers.
6. Der Resource-Adapter startet die Ausführung der Prozess-Instanz.
7. Der Resource-Adapter beendet die Ausführung der Prozess-Instanz, und bekommt ein OutContainer als Rückgabewert.
8. Der Resource-Adapter startet die Abbildung von einem OutContainer zu einer *IndexedRecord*-Instanz.
9. Der Resource-Adapter beendet den Abbildungsprozess.
10. Das Bean oder die Java-Applikation ruft die Methode *Connection.Close()* auf, um die Verbindung zu dem MQWF-Server zu beenden. Im Managed-Fall wird nur die Verbindung auf Applikationsebene beendet, während im Non-Managed-Fall die physikalische Verbindung zu dem MQWF-Server beendet wird.
11. Die Verbindung wurde beendet.
12. Beendigung der Programmausführung.



Managed-Applikationsszenario Non-Managed-Applikationsszenario

Abbildung 8.1: Die Messpunkte für die Messungen

8.4 Messergebnis

8.4.1 Der Gesamtzeitvergleich

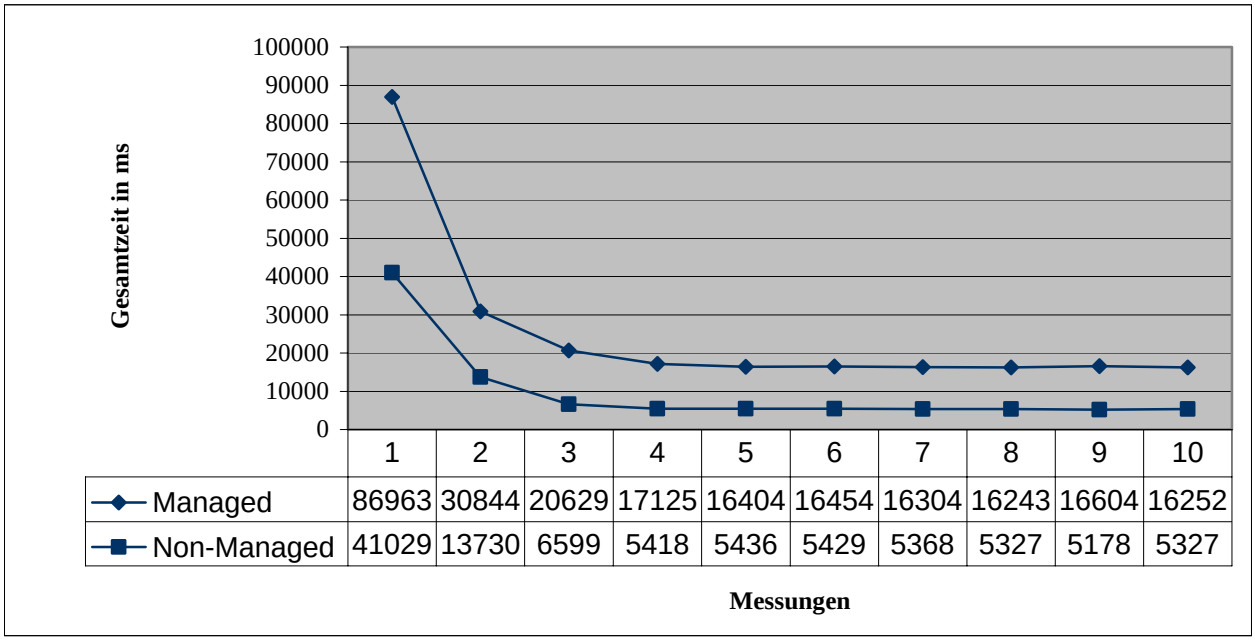
Die Gesamtzeit umfasst die Zeit zwischen Messpunkt 1 und Messpunkt 12, nämlich die Zeit, die für die Ausführung der Anwendung oder die Ausführung der Java-Applikation gebraucht wird. Die Zeiten werden im Messprogramm gemessen.

Um die Ergebnisse vergleichen zu können, wurden folgende Bedingungen erfüllt: erstens, errichten die Anwendung und die Java-Applikation mit dem MQWF-Server eine Verbindung, wobei immer mit dem gleichen Benutzernamen und dem gleichen Passwort eingeloggt wird. Zweitens, greifen sie immer auf dieselben Prozess-Templates zu. Drittens, benutzen sie stets die gleichen Daten, um eine Prozess-Instanz auszuführen.

Für den Managed- und Non-Managed-Fall werden 5 Messdurchläufe gestartet, wobei in jedem Messdurchlauf das Beispielprogramm 10 mal nacheinander ausgeführt wird. Um die gleiche Umgebung für die Ausführung des Programms zu schaffen, wird der WebSphere-Server und der MQWF-Server für jeden Messdurchlauf neu gestartet.

Um das letztendliche Messergebnis für einen Messpunkt zu erhalten, nimmt man den Medianwert der gemessenen Daten an den gleichen Messpunkten im Programm, die aus 5 verschiedenen Messdurchläufen stammen.

Die Abbildung 8.2 zeigt die Gesamtzeit von 10 Messungen im Managed-Fall und Non-Managed-Fall. Die dargestellten 10 Messungen beschreiben die Zusammenfassung von 5 Messdurchläufen. Beide Kurven zeigen einen ähnlichen Verlauf, was man an der anfangs starken Reduzierung der Gesamtzeiten und der anschließenden Abflachung beider Kurven in einem fast konstant endenden Verlauf, erkennen kann. Ein Grund liegt vermutlich darin, dass bei der ersten Ausführung einige Daten komplett neu in den Hauptspeicher geladen werden müssen, und noch nicht gecached sind. Im Managed-Fall liegt ein weiterer Grund darin, dass die physikalische Verbindung zu dem MQWF-Server nur bei der ersten Ausführung erzeugt werden muss. Die Gesamtzeiten im Managed-Fall sind allgemein länger als im Non-Managed-Fall, weil die Initialisierung einer J2EE-Applikation im Managed-Fall lange dauert, was später noch aufgeklärt wird.



	1	2	3	4	5	6	7	8	9	10
Managed	86963	30844	20629	17125	16404	16454	16304	16243	16604	16252
Non-Managed	41029	13730	6599	5418	5436	5429	5368	5327	5178	5327

Abbildung 8.2: Die Gesamtzeitmessungen für den Managed- und Non-Managed-Fall

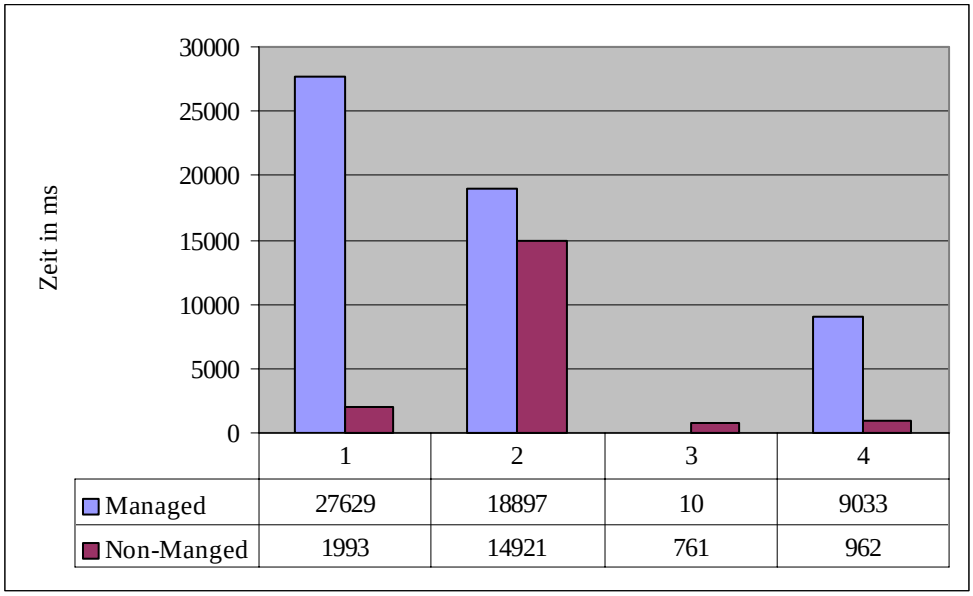
8.4.2 Einzelne Zeitvergleiche

Ein großer Unterschied zwischen dem Managed-Fall und dem Non-Managed-Fall besteht darin, dass im Managed-Fall ein Benutzer sich nur beim ersten Mal auf dem MQWF-Server einloggen muss, und für den eingeloggten Benutzer muss der Programm-Execution-Agent nur ein Mal gestartet werden. Im Non-Managed-Fall muss sich der Benutzer jedes Mal auf dem MQWF-Server einloggen und nach der Ausführung der Prozess-Instanz wieder ausloggen. Nach dem Einloggen des Benutzers muss der Programm-Execution-Agent jedes Mal neu gestartet und vor dem Ausloggen des Benutzers gestoppt werden. Weil der Programm-Execution-Agent immer mit einem eingeloggten Benutzer verbunden ist.

Vergleich für die erste Ausführung

Hier werden zuerst die Messergebnisse von der ersten Ausführung des Programms für den Managed-Fall und für den Non-Managed-Fall verglichen. Die Abbildung 8.3 zeigt die einzelnen Zeitvergleiche für die einzelnen Phasen im Managed- und Non-Managed-Fall.

Phase 1 umfasst die Zeit zwischen Messpunkt 1 und Messpunkt 2, nämlich die Zeit zwischen dem Start des Anwendungsprogramms und dem Aufruf der Methode *Connection.getConnection()* im Resource-Adapter. Im Managed-Fall wird in der Phase 1 zuerst die J2EE-Anwendung von dem WebSphere-Server initialisiert. Dann findet der Client das Session-Bean im JNDI-Name-Space. Der Client liest die Daten aus der Properties-Datei, und übergibt dem Bean diese Daten. Im Non-Managed-Fall wird das Programm in der Phase 1 mit dem Befehl *java* gestartet. Das Programm liest die Daten aus der Properties-Datei, bevor es die Methode *Conection.getConnection()* aufruft.



	1	2	3	4
Managed	27629	18897	10	9033
Non-Manged	1993	14921	761	962

Abbildung 8.3: Einzelne Zeitvergleiche für die erste Ausführung

Die Abbildung 8.3 zeigt den Zeitunterschied zwischen dem Managed- und Non-Managed-Fall in Phase 1. Im Managed-Fall dauert es fast 14 mal so lang wie im Non-Managed-Fall. Ein wesentlicher Grund dafür ist, dass im Managed-Fall die Initialisierung der J2EE-Anwendung von dem WebSphere-Server sehr viel Zeit in Anspruch nimmt. Zuerst werden die Befehlszeilenparameter von dem Applikationsserver verarbeitet. Dann wird die Umgebung des J2EE-Application-Clients initialisiert. Außerdem muss der Client im Managed-Fall das EJB-Objekt im JNDI-Namespace finden, eine EJB-Objekt-Instanz erzeugen und die Methode der EJB-Objekt-Instanz remote aufrufen, während im Non-Managed-Fall alles in einem Programm passiert.

Die Phase 2 zeigt die Zeit, die benötigt wird, um eine Verbindung zu dem MQWF-Server zu bekommen. Die Messpunkte 2 und 3 wurden hier gemessen. In dieser Phase ruft das Bean oder die Java-Applikation die Methode *Connection.getConnection()* im Resource-Adapter auf, und der Resource-Adapter benutzt den Benutzernamen und das Passwort zum Einloggen

auf dem MQWF-Server. Nach erfolgreichem Einloggen liefert der Resource-Adapter dem Bean oder der Java-Applikation die Verbindung auf Applikationsebene zurück. In dieser Phase dauert es im Managed-Fall etwas länger als im Non-Managed-Fall. Das bedeutet, dass es länger dauert, eine neue Verbindung von einem Applikationsserver zu einem MQWF-Server zu erzeugen, als eine neue Verbindung von einer Java-Applikation zu einem MQWF-Server.

Zwischen Messpunkt 3 und 4 wird hauptsächlich die *Interaction*-Instanz erzeugt und die Methode *Interaction.execute()* aufgerufen. Es zeigt keinen großen Unterschied zwischen dem Managed- und Non-Managed-Fall. Deswegen wird es hier nicht genau beschrieben.

Die Messpunkte 4, 5, 6, 7, 8 und 9 wurden gemessen, um das Zeitverhalten des Resource-Adapters in Abhängigkeit von Datenvolumina zu zeigen. Dies wird später näher erklärt.

In der Phase 3 wird das Bean oder die Java-Applikation die Verbindung zu dem MQWF-Server beenden. Die Phase umfasst die Zeit zwischen Messpunkt 10 und 11. Im Managed-Fall wird nur die Verbindung auf Applikationsebene beendet. Die physikalische Verbindung wird von dem WebSphere-Server zum Connection-Pool hinzugefügt. Das bedeutet, dass der Benutzer nicht aus dem MQWF-Server ausgeloggt werden muss. Im Non-Managed-Fall muss der Benutzer der Java-Applikation noch aus MQWF-Server ausgeloggt werden. Die Phase 3 in der Abbildung 8.3 zeigt den Zeitunterschied zwischen dem Managed- und Non-Managed-Fall. Im Non-Managed-Fall dauert es länger als im Managed-Fall, da der Programm-Execution-Agent gestopped und der Benutzer noch ausgeloggt werden muss. Hier wird der Vorteil des Connection-Pooling im Managed-Fall gezeigt. Aber die Zeit, die für den Abbau der physikalischen Verbindung im Managed-Applikationsszenario gespart hat(751ms), ist relativ gering.

Die Phase 4 umfasst die Dauer zwischen dem Messpunkt 11 und dem Messpunkt 12. Konkret, ist das die Dauer von der Verbindungsbeendigung des Beans oder der Java-Applikation bis zum Ende der Ausführung der Anwendung. In dieser Phase übergibt das Bean dem Client im Managed-Fall die vom Resource-Adapter zurückgelieferten Daten, und der Client speichert diese Daten in einer Properties-Datei. Im Non-Managed-Fall speichert die Java-Applikation die zurückgelieferten Daten auch in einer Properties-Datei. Die Abbildung 8.3 zeigt, dass es beim Managed-Fall fast 10 mal so lang wie beim Non-Managed-Fall gedauert hat. Ein Grund dafür ist, dass im Managed-Fall der Client mit dem Bean remote kommuniziert, während im Non-Managed-Fall alles lokal passiert.

Vergleich für die stabilisierten Zustände

Wie in der Abbildung 8.2 illustriert wird, bleibt der, nach 4 aufeinanderfolgenden Ausführungen, gezeigte Verlauf der Gesamtzeit für die Ausführung der Anwendung fast konstant. Deswegen wird der Medianwert von den Messungen 5 bis 10 verwendet, um das Zeitverhältnis zwischen dem Managed-Fall und dem Non-Managed-Fall weiter zu vergleichen. Das Ergebnis zeigt die Abbildung 8.4.

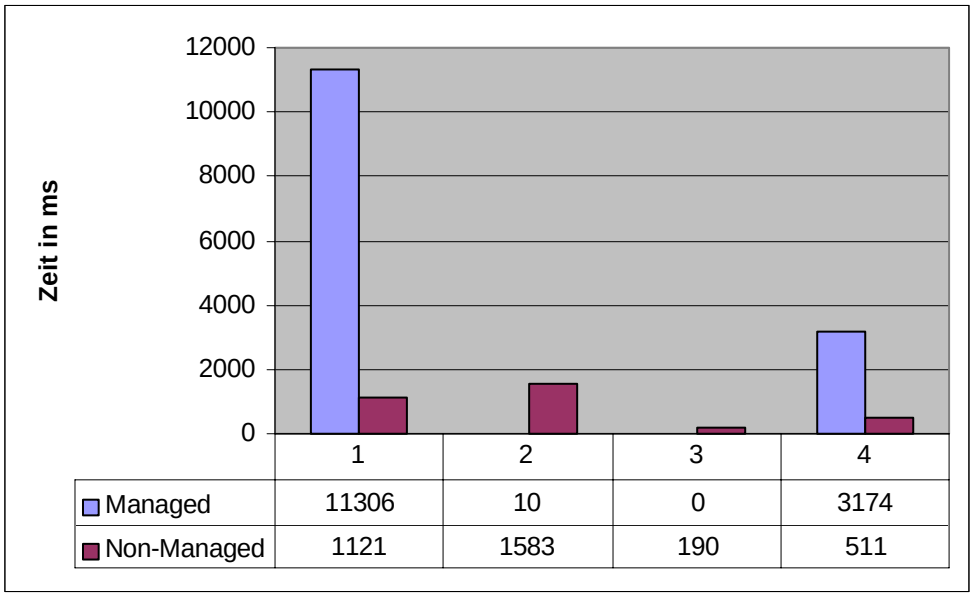


Abbildung 8.4: Einzelne Zeitvergleiche für die stabilisierten Zustände

Wie beim ersten Vergleich in Abbildung 8.3, zeigt Abbildung 8.4 den großen Zeitunterschied zwischen dem Managed- und dem Non-Managed-Fall in der Phase 1 und der Phase 4. Das erklärt auch, warum im Managed-Fall allgemein mehr Zeit als im Non-Managed-Fall gebraucht wird, eine Anwendung auszuführen, wie es Abbildung 8.2 zeigt.

Obwohl die physikalischen Verbindungen im Managed-Fall weiter verwendet werden können, bleibt der Resource-Adapter im Managed-Fall immer uneffizienter als im Non-Managed-Fall. Der Resource-Adapter spart die Zeit für die Erzeugung und das Beenden einer physikalischen Verbindung im Managed-Fall, aber die Initialisierung einer J2EE-Anwendung und die Remote-Kommunikation zwischen einem Client und einem Bean braucht zusätzlich noch viel Zeit, im Vergeich zum Non-Managed-Fall.

In Abbildung 8.4 sehen wir, dass, die Zeit, die der Resource-Adapter im Managed-Fall bei der Verbindungserzeugung und dem Verbindungsbeenden spart (1763ms), im Vergleich zu der Zeit, die im Managed-Fall als im Non-Managed-Fall in Phase 1 und 4 mehr gebraucht wird(12848ms), sehr gering ist. Hier erkennt man, dass die J2EE-Anwendung allgemein uneffizienter als die Java-Applikation ist.

In der Phase 2 zeigt die Abbildung 8.4 einen großen Unterschied zu der Abbildung 8.3. Die Abbildung 8.4 zeigt, dass im Managed-Fall nur 10ms benötigt wurde, um eine Verbindung zu dem MQWF-Server zu erstellen. Das liegt daran, dass nur die Verbindung auf Applikationsebene erstellt wird und die physikalische Verbindung vom Connection-Pool auf dem WebSphere-Server geholt und wieder verwendet wird. Im Non-Managed-Fall beträgt die gemessene Dauer 1583ms, um sich neu auf dem MQWF-Server einzuloggen und den Programm-Execution-Agent für den Benutzer zu starten. Hier sehen wir, dass wegen der Unterstützung des Connection-Poolings des Applikationsservers im Managed-Fall die Zeiten für die Erzeugung und das Beenden einer physikalischen Verbindung gespart werden.

8.4.3 Abhängigkeit von Datenvolumina

Um das Zeitverhalten eines Resource-Adapters zu untersuchen, ist die Abhängigkeit von Datenvolumina auch wichtig. Ein Resource-Adapter benutzt die Eingabedaten, um den Input-Container einer Prozess-Instanz zu initialisieren. Die Ausführung der Prozess-Instanz wird von dem MQWF-Server durchgeführt. Nach der Ausführung einer Prozess-Instanz liest der Resource-Adapter die Daten von dem Output-Container der Prozess-Instanz, und packt die Daten in eine resource-adapter-spezifische Datenstruktur ein, was den Abbildungsprozess beendet.

Für die oben genannten drei Phasen wurden die Messpunkte 4, 5, 6, 7, 8 und 9 gemessen. Die Zeit zwischen den Messpunkten 4 und 5 wird für die Initialisierung gebraucht. Zwischen den Messpunkten 6 und 7 wird die Zeitdauer für die Ausführung der Prozess-Instanz gemessen. Zwischen den Messpunkten 8 und 9 wird die Abbildung (Mapping) von einem Container zu einer resource-adapter-spezifischen Datenstruktur im Resource-Adapter durchgeführt. Zwischen den Messpunkten 5 und 6, 7 und 8 wird kaum Zeit gebraucht, weil nur wenige Zuweisungen vorkommen. Die Messungen wurden im Managed-Applikationsszenario durchgeführt.

Um den Input-Container zu initialisieren, muss der Resource-Adapter jedes Container-Element mit dem vollständigen Namen und dem Wert initialisieren. Zum Beispiel, benutzt der Resource-Adapter die Funktion *inContainer.setString(name, value)*, um ein Container-Element, dessen Datentyp *String* ist, zu initialisieren. Deswegen ist die Anzahl der Container-Elemente sehr wichtig, um die Zeitdauer der Initialisierung zu messen. Für die Abbildung von einem Container zu einer resource-adapter-spezifischen Datenstruktur spielt die Anzahl der Container-Elemente ebenso eine wichtige Rolle. Wegen der oben genannten Gründe wurde das folgende Verfahren für diese Messungen eingesetzt.

Für das Messen wird ein Workflow mit einer Aktivität, mit der ein Java-Programm verbunden ist, verwendet. Das Java-Programm liest die Daten von dem InputContainer und kopiert die Daten in den OutputContainer. Eine Basis-Datenstruktur mit 50 Daten-Membern wird im Build-Time definiert. Zum Messen verwendet der Workflow eine Datenstruktur mit verschiedenen Größen, die von der Basisdatenstruktur erzeugt werden. Zum Beispiel, können Datenstrukturen mit 100 oder 150 Daten-Membern erzeugt werden.

MQSeries-Workflow unterstützt die Datenstruktur nur mit maximal 512 Daten-Membern [IBMWFBT]. Hier wurde die Datenstruktur mit bis 500 Daten-Membern zum Messen verwendet.

Initialisierung

Die Initialisierungprozess passiert sowohl auf dem Resource-Adapter, als auch auf dem MQWF-Server. Auf dem Resource-Adapter werden die Daten zuerst von einer Datenstruktur ausgepackt. Für jeden Datentyp verwendet der Resource-Adapter verschiedene Funktionen, um den Container zu initialisieren. Auf dem MQWF-Server wird das entsprechende Container-Element initialisiert. Die Zeit wird hier für die Messpunkte 4 und 5 gemessen.

Die Abbildung 8.5 zeigt das Messergebnis für die Zeitdauer der Initialisierung des InputContainers. Von der Kurve sehen wir, dass die absolute Zeitdauer ändert sich nicht so stark. Die Zeit für die Initialisierung nur ca. 120ms beträgt, wenn die Anzahl der Container-Elemente von 50 auf 500 steigt. Bis auf ein paar regelmäßige Abweichungen ist der Verlauf fast linear ansteigend. Das ist darauf zurückzuführen, dass die Messergebnisse in einem sehr schmalen Zeitbereich im Millisekundenbereich gemessen wurden. Unregelmässigkeiten kommen daher zu Stande, dass die Millisekundenangabe der Funktion *System::getCurrentTimeMillis()* in einem Zeitraster von 10 ms gegeben sind.

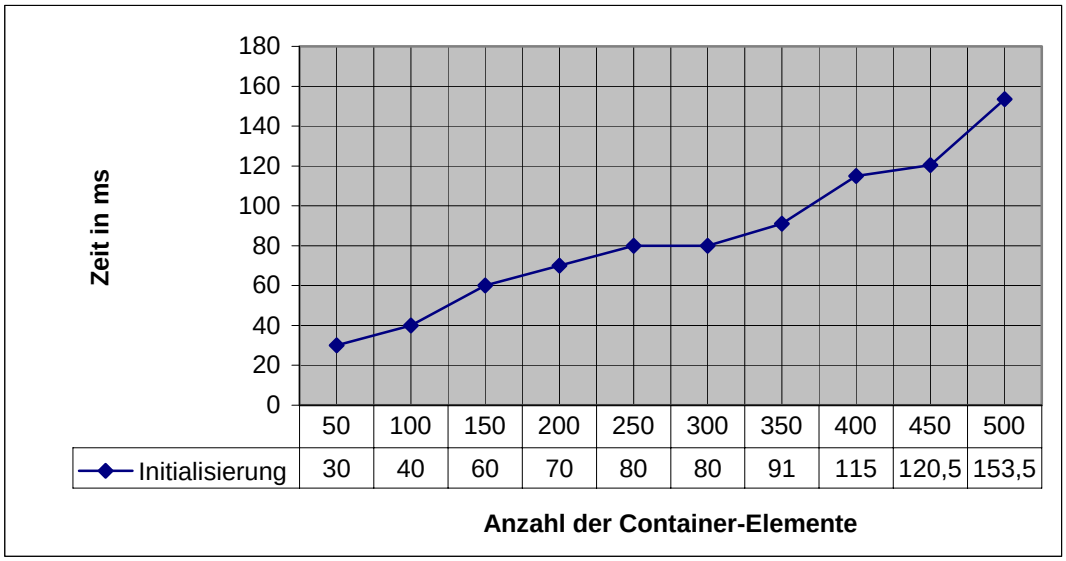


Abbildung 8.5: Die Zeitdauer für die Initialisierung in Abhängigkeit der Datenmenge

Abbildung

Die Abbildung von einem Output-Container zu einer Datenstruktur(*IndexedRecord*-Instanz) geschieht auf dem Resource-Adapter. Die Zeitdauer umfasst die Zeit zwischen den Messpunkten 8 und 9. Der Resource-Adapter liest die Daten aus dem Output-Container und packt sie in eine *IndexedRecord*-Instanz.

Die Abbildung 8.6 zeigt die Kurve der Zeitdauer für die Abbildung in Abhängigkeit der Datenmenge. Das Diagramm zeigt, dass in Abhängigkeit von der Anzahl der Container-Elementen die Kurve fast linear ansteigt. Die verbrauchte Zeit steigt jedoch nur von 20ms auf 175ms, mit einem Anstieg der Container-Elemente- Anzahl von 50 auf 500. Ungenauigkeiten, die die Kurve in Abbildung 8.6 aufweist, beruhen auf der gleichen Tatsache, die bereits für die Kurve in Abbildung 8.5 gegeben ist.

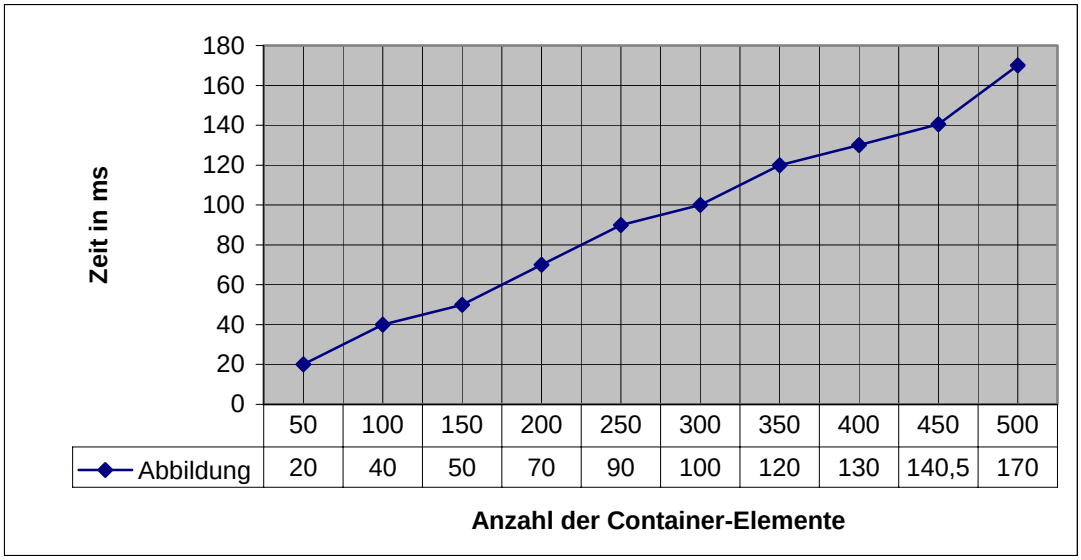


Abbildung 8.6: Die Zeit für die Abbildung in Abhängigkeit der Datenmenge

Ausführung

Der Resource-Adapter benutzt die Workflow-Java-API-Funktion, um eine Prozess-Instanz zu erzeugen und auszuführen. Die Ausführung der Prozess-Instanz wird von dem MQWF-Server durchgeführt. Hier werden die Messpunkte 8 und 9 gemessen.

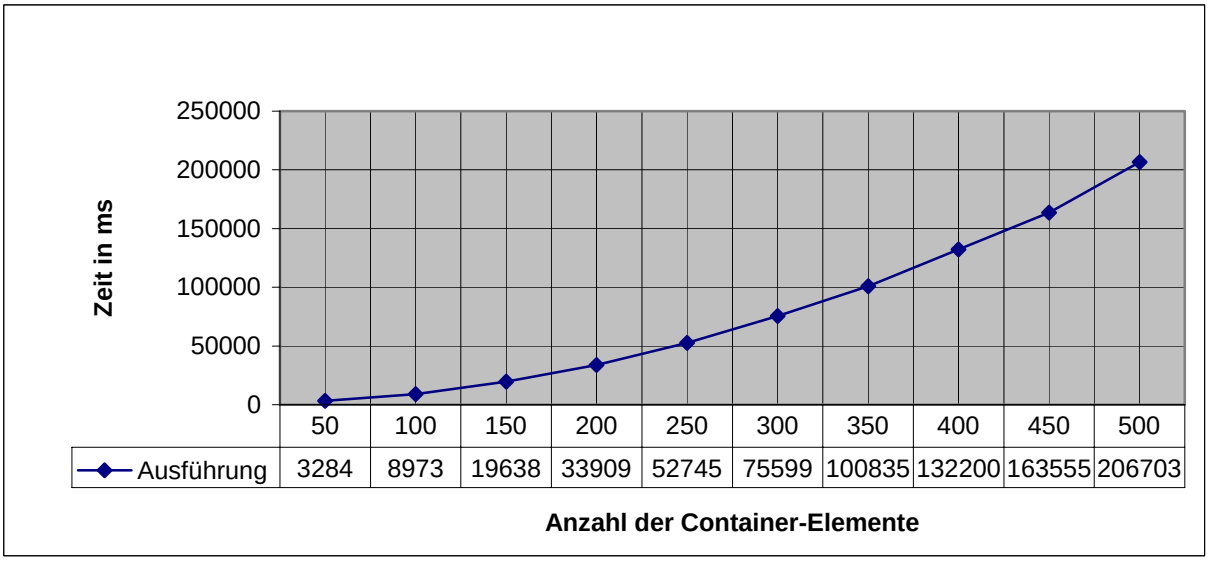


Abbildung 8.7: Die Zeit für die Ausführung in Abhängigkeit der Datenmenge

Die Abbildung 8.7 zeigt die Zeit für die Erzeugung und für die Ausführung einer Prozess-Instanz in Abhängigkeit der Datenmenge. Die gezeigte Kurve steigt polynomiell an. Der Grund dafür liegt an dem einzigen Aktivitätsprogramm in der Prozess-Instanz.

Im Vergleich zu der Zeitdauer für die Ausführung einer Prozess-Instanz ist die Dauer für die Initialisierung eines Containers und für den Abbildungsprozess sehr unbedeutend. Die Auswirkung der Datengröße beeinflusst hauptsächlich das Zeitverhalten der Ausführung der Prozess-Instanz. Das Datenvolumen spielt also für die Initialisierungs- und Abbildung-Phase kaum eine Rolle.

9 Zusammenfassung und Ausblick

Das Ziel dieser Diplomarbeit ist es, einen Resource-Adapter für MQSeries-Workflow in der J2EE-Connector-Architektur prototypisch zu implementieren. Die standardisierte Connector-Architektur definiert die Interfaces, die ein Resource-Adapter und ein Applikationsserver implementieren muss. Die System-Verträge sind zwischen einem Resource-Adapter und einem Applikationsserver definiert, damit der Applikationsserver dem Resource-Adapter die Basis-Dienste (Sicherheit-Management, Transaction-Management und Connection-Management) anbieten kann. Der Resource-Adapter bietet auch die Interfaces für den Client, mit denen der Client den Resource-Adapter verwenden und auf den MQWF-Server zugreifen kann.

Ein Vorteil der J2EE-Connector-Architektur besteht darin, dass ein Applikationsserver nur einmal erweitert werden muss, um mit verschiedenen Enterprise-Information-Systems interagieren zu können, und ein Enterprise-Information-System muss auch nur einmal die J2EE-Connector-Architektur implementieren, um mit verschiedenen Applikationsservern zu interagieren.

In dieser Diplomarbeit wurde der WebSphere-Server als Applikationsserver verwendet. Wegen der Einschränkung des WebSphere-Servers und des MQWF-Servers wurden nicht alle Eigenschaften, die in der Connector-Architektur definiert werden, in diesem Resource-Adapter implementiert. Zum Beispiel unterstützt der Resource-Adapter keine Transaktionen, weil MQSeries-Workflow keine Transaktionen anbietet. Der WebSphere-Server unterstützt nur Component-Managed-Sign-On, deswegen wird hier in diesem Resource-Adapter auch nur Component-Managed-Sign-On implementiert, nicht Container-Managed-Sign-On.

Auf Basis der J2EE-Connector-Architektur unterstützt ein Resource-Adapter Managed- und Non-Managed-Applikationsszanarien. Im Managed-Applikationsszenario läuft der Resource-Adapter und die J2EE-Anwendung auf dem Applikationsserver. Zum Beispiel läuft ein Enterprise-Java-Bean in dem Container eines Applikationsservers. Im Non-Managed-Applikationsszenario läuft der Resource-Adapter nicht auf dem Server. Er wird von den Applikationen (z.B. Java-Applikation) als Bibliothek verwendet.

Im Managed-Applikationsszenario bietet der WebSphere-Server dem Resource-Adapter den Mechanismus des Connection-Poolings. Mit der Unterstützung des Connection-Poolings muss der Resource-Adapter für einen Benutzer nur ein Mal eine physikalische Verbindung zum MQSeries-Workflow-Server erzeugen. Falls ein Resource-Adapter intern kein Connection-Pooling implementiert, muss der Resource-Adapter im Non-Managed-Applikationsszenario für einen Benutzer jedes Mal eine neue physikalische Verbindung erzeugen und beenden, wenn der Benutzer auf MQSeries-Workflow zugreifen möchte. Im Vergleich zu dem Non-Managed-Applikationsszenario spart der Resource-Adapter im Managed-Applikationsszenario die Zeit für die Erzeugung und das Beenden der physikalischen Verbindung. Im Managed-Applikationsszenario bietet der Applikationsserver zusätzlich noch die Dienste für das Sicherheits- und Transaktionsmanagement, während im Non-Managed-Applikationsszenario der Applikationsclient dafür zuständig ist.

In dieser Diplomarbeit wurde auch eine J2EE-Anwendung und eine Java-Applikation für die Managed- und Non-Managed-Applikationsszenarien zum Testen implementiert. Das Zeitverhalten des Resource-Adapters wurde im Managed- und Non-Managed-Fall untersucht. Das Ergebnis zeigt, dass obwohl der Applikationsserver im Managed-Applikationsszenario Connection-Pooling unterstützt, der Resource-Adapter im Managed-Applikationsszenario allgemein uneffizienter als im Non-Managed-Applikationsszenario bleibt. Ein wesentlicher Grund dafür ist, dass im Managed-Applikationsszenario viel Zeit für die Initialisierung einer J2EE-Anwendung gebraucht wird. Dazu gehören die Initialisierung eines J2EE-Anwendungsclients, die Erzeugung einer EJB-Objekt-Instanz, die Remote-Kommunikation zwischen einem Client und einem EJB usw..

Die Zeitdauer des Resource-Adapters wurde auch in Abhängigkeit von der Datengröße der Eingabe- und Ausgabe-Daten untersucht. Das Ergebnis zeigt, wenn die Datenmengen vergrößert werden, dass die Zeit, die der Resource-Adapter für die Initialisierung des Input-Containers und für die Abbildung des Output-Containers auf eine resource-adapter-spezifische Datenstruktur benötigt, steigt, jedoch sehr gering, im Vergleich zu der Zeit, die für die Ausführung einer Prozess-Instanz gebraucht wird.

Dieser Resource-Adapter ist eine prototypische Implementierung, die weiter verbessert werden kann. Zum Beispiel, kann ein Resource-Adapter einen internen Connection-Pool-Mechanismus implementieren, damit im Non-Managed-Applikationsszenario die Kosten für die Verbindungserzeugung und das Verbindungsbeenden gespart werden können.

Das Common-Client-Interface ist mit dem JDBC-API vergleichbar. Das JDBC-API liefert dem Client ein *ResultSet* zurück. Dieser Resource-Adapter wurde so implementiert, dass er dem Client einen *IndexedRecord* zurückliefert, welcher auf *java.util.List* basiert, und eine resource-adapter-spezifische Datenstruktur repräsentiert. Es ist auch denkbar, dass der Resource-Adapter dem Client eine standardisierte Datenstruktur wie den, im JDBC-API verwendeten *ResultSet* zurückliefern kann.

Dieser Resource-Adapter wurde so implementiert, dass er dem Client nach der Ausführung einer Prozess-Instanz alle Elemente eines Output-Containers zurückliefert. Das ist ein Nachteil, wenn der Client nicht alle Informationen vom Output-Container braucht. Deswegen kann der Resource-Adapter auch so verbessert werden, dass er dem Client ermöglicht, nur die gewünschten Daten zu erhalten.

Literatur:

[CONSPE10]: J2EE Connector Architecture Specification, Final release 1.0

[IBMWFCA]: IBM. IBM MQSeries Workflow, Concepts and Architecture, Version 3.3.2

[WASINFO]: IBM. IBM WebSphere InfoCenter für Application Server, Advanced Edition

[WRM95]: WfMC. The Workflow Management Coalition Specification, Workflow Management Coalition, The Workflow Reference Model, Document Number TC00-1003, Issue 1.1, January,1995

[IBMWFBT]: IBM. IBM MQSeries Workflow, Getting Started with Buildtime, Version 3.3.2

[IBMWFPG]: IBM. IBM MQSeries Workflow, Programming Guide, Version 3.3.2

[J2EEINTRO]: Monica Pawlan. Introduction to the J2EE Platform. March 2001. http://developer.java.sun.com/developer/technicalArticles/J2EE/Intro/index.html

[J2EERA]: Jennifer Rodoni. The Java™ 2, Enterprise Edition (J2EE™) Connector Architecture's Resource Adapter. December 2001. http://developer.java.sun.com/developer/technicalArticles/J2EE/connectorclient/resourceadapter.html

[FLDR] Frank Leymann und Dieter Roller. Production Workflow, Concepts and Techniques. Prentice Hall PTR, 2000.