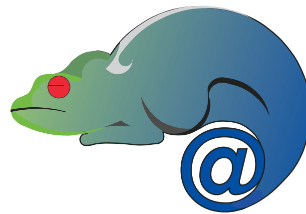
ADAPT

IST-2001-37126

Middleware Technologies for Adaptive and Composable Distributed Components

Prototype of the Transactional Engine



Deliverable Identifier:	D4
Delivery Date:	02/29/2004
Classification:	Public Circulation
Authors:	F. Pérez Sorrosal, M. Patiño Martínez, R. Jiménez Peris
Document version:	1.0, 02/27/2004
Contract Start Date:	1 September 2002
Duration:	36 months
Project Coordinator:	Universidad Politécnica de Madrid (Spain)
Partners:	Università di Bologna (Italy), ETH Zürich (Switzerland), McGill University (Canada), Università di Trieste (Italy), University of Newcastle (UK), Arjuna Technologies Ltd.(UK)

Project funded by the European Commission under the Information Society Technologies Programme of the 5th Framework (1998-2002)

Contents

1 Introduction **3**

2 The J2EE Activity Service Specification **3**

3 J2EE Activity Service Architecture **4**

4 A High Level Service for the Open Nested Transaction Model **4**
4.1 The ONT Model 5
4.2 Using the ONT Model 6

5 ONT HLS Implementation **10**
5.1 Deploying the Activity Service and an HLS 10

6 Integration with Web Services **10**
6.1 A Conversational Web Service 11

A Software Distribution **13**

1 Introduction

In this document we describe the advanced transactional support provided in the context of the ADAPT project for basic services (BSs). The transactional support is provided by a transactional engine that supports advanced transaction models. These transaction models arise due to the increasing complexity of applications that need more complex transactional semantics than those provided by the classical short-lived ACID transactions. The composition and coordination of web services with transactional semantics is an example of those applications.

There are many advanced transaction models in the literature; however, no single transaction model satisfies all types of applications. A discussion on this topic can be found in deliverable D5: Transactional Support [JP03]. The OMG *Activity Service* (AS) [OMG02] defines a generic framework for implementing advanced transaction models in CORBA. The *J2EE Activity Service specification* [Sun03] is the adaptation of the CORBA Activity Service to the J2EE environment.

The prototype of the transactional engine implemented in this deliverable is an implementation of the J2EE Activity Service specification (ASS) draft 0.1, June 2003. As part of this deliverable the *open nested transaction model* (ONT) has also been implemented as a proof of concept along with a simple application based on this model. The rest of the document describes the J2EE AS specification, the ONT model and the example that uses ONTs.

2 The J2EE Activity Service Specification

The Activity Service provides an abstract unit of work, *activity*, that may or may be not transactional. An activity may encapsulate a JTA transaction [Sun99a] or be encapsulated by a JTA transaction. Activities may be nested. Figure 1 (borrowed from [HLR+03]) shows a complex structure of activities. The dotted ellipses represent activity boundaries, whereas the solid ellipses are transaction boundaries. Activity A1 contains two nested transactions, while activities A2, A4 and A5 contain no transaction. Activity A3 is more involved; it contains a transaction, which again contains one activity (A3') that contains a transaction. Activities A1 and A2 are sequential, while A3 and A4 are executed in parallel.

Activities are explicitly demarcated. They are created, executed and completed, producing an *outcome*. Demarcation points are communicated to registered entities (*actions*) through *signals*, which are produced by *signalsets*. The signalset is a finite state machine that accepts the outcomes of actions as input. The signalset may use that outcome to determine the next signal to send. The next signal will not be produced until the previous one has been sent to all registered actions. Based on this model, the J2EE ASS defines the interfaces and behaviour of an Activity Service such that using those interfaces, advanced transaction models can be implemented. A transaction model defines the signals that may be produced during the lifetime of an activity, the outcomes and the state transitions (signalset) produced as signals are consumed.

For instance, the two phase commit protocol may be implemented using the AS [HLR+03]. The signals for this protocol would be *prepare, commit* and *abort*. The actions registered with the signalset are the protocol participants. Once the transaction (activity) finishes, the signalset produces the *prepare* signal. This signal is communicated to all participants (actions) one by one. Each action may produce a different outcome (the vote), which will determine the next signal to be produced. Assuming that all participants vote *yes*, the next signal (*commit*) will be sent to all actions.

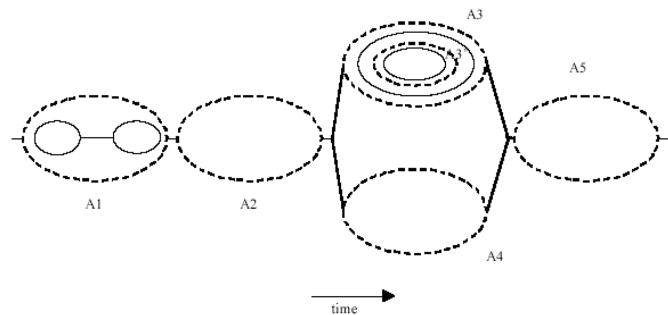


Figure 1: Activities and transactions

3 J2EE Activity Service Architecture

There are two main components in the AS architecture: *High Level Services* (HLS) and the Activity Service (AS) itself (Figure 2 from [Sun03]).

A High Level Service defines a specific transaction model. Applications use a HLS to demarcate their activities; they do not interact with the AS. A HLS interacts with the AS, which distributes the signals of the HLS. The ASS does not define implementations of HLS. It just defines the interfaces a HLS must implement and the interfaces a HLS may invoke on the underlying AS.

The Activity Service is a low-level general purpose engine for registering and triggering events with the notion of activity scopes (contexts). The AS provides two interfaces, `UserActivity` and `ActivityManager`. The `ActivityManager` interface is used by containers. Methods of the `UserActivity` interface are used by a HLS to control and demarcate the scope of an activity. These methods also allow pluggable coordination. The AS manages HLS's contexts, including JTS contexts [Sun99b] and implicitly propagates them with remote requests.

The separation of these two elements allows the plugging of different transaction models into an AS implementation. New transaction models are added just by providing the corresponding HLS. The AS does not require any changes, since it does not contain details about any transaction model.

The interaction between a HLS and the AS is shown in Figure 3 [Sun03]. A HLS provides a `ServiceManager` to the AS through a `UserActivity` instance. The ServiceManager is used by the AS to obtain the HLS signalset. The `ActivityCoordinator` uses the signalset to produce signals and send them to actions. It also returns the outcomes of the actions to the signalset.

4 A High Level Service for the Open Nested Transaction Model

This Section describes the *open nested transaction model* (ONT) and an HLS that implements this model. The ONT HLS implements the required interfaces (`ServiceManager, SignalSet, Action`) to communicate with the underlying AS implementation and offers to applications a simple interface (`UserOpenNested`) in order to use ONTs. These interfaces are defined in [Sun].

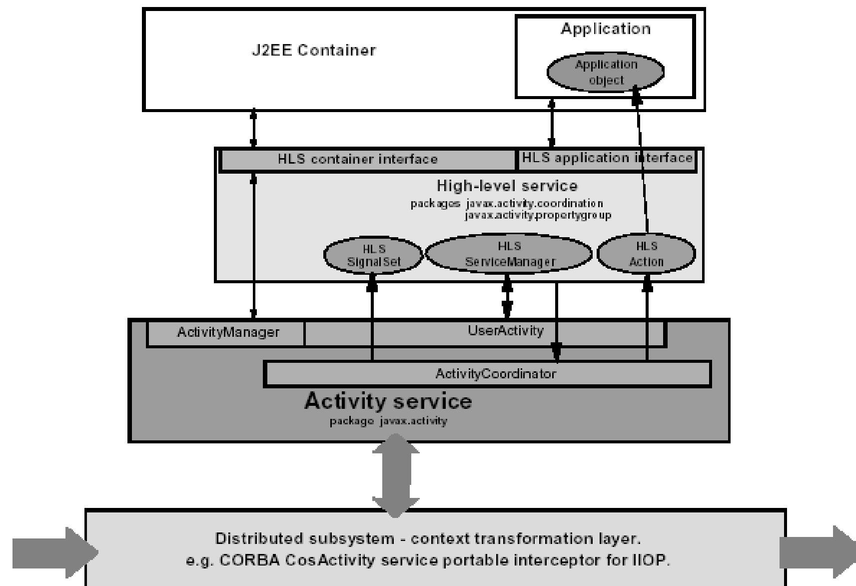


Figure 2: J2EE Activity Service architecture

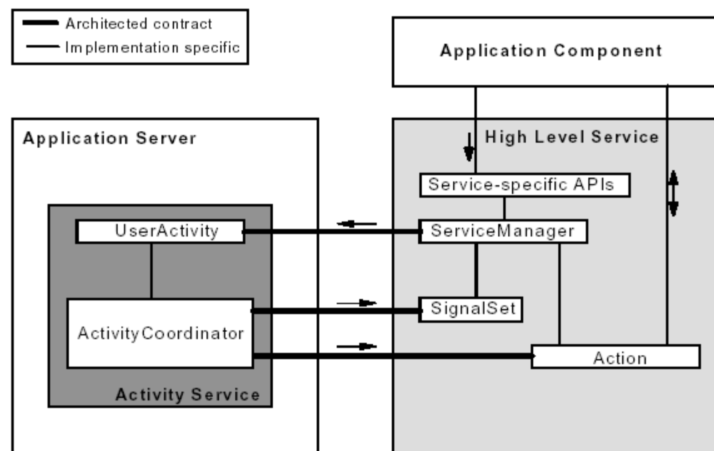


Figure 3: Interaction between the AS and a HLS

4.1 The ONT Model

In the ONT model each activity represents an atomic unit of work, a JTA transaction. Any activity creation implies the creation of an associated top-level transaction. Activities may contain any number of nested activities, which may again contain other nested activities organized into a hierarchical tree. During the execution of a nested activity, the parent transaction is suspended. The parent transaction will resume, when the nested activity finishes.

The ONT model was originally defined in [WS92]. We will follow a variation of the ONT defined in [Sun03]:

1. An ONT activity cannot succeed unless all of its children have completed. Since an ONT activity is transactional, success means that the associated transaction committed.

2. When an ONT activity completes with failure, all of its children that are still in an active state are completed with failure. Completing with failure means that the associated transaction aborts.

3. When an ONT activity completes with failure, all of its children that previously completed with success will be compensated (in reverse order of completion), if compensating actions have been defined for them. The behavior of the compensation action is defined by the application.

4.2 Using the ONT Model

We have implemented a sample application based on the travel agency example in order to test the implementation of the ONT HLS. The application provides a booking service for a package holiday consisting of a flight and a hotel. For this prototype, we model both the airline and the hotel services as stateless session beans. The user just inputs a name and the number of tickets and rooms needed through a web page (Figure 4). If the current availability of either tickets or rooms is less than the number requested, the reservation will fail and all its effects must be undone (compensated).

The web page shows the current availability of flight seats and hotel rooms in order to check the invocation results.

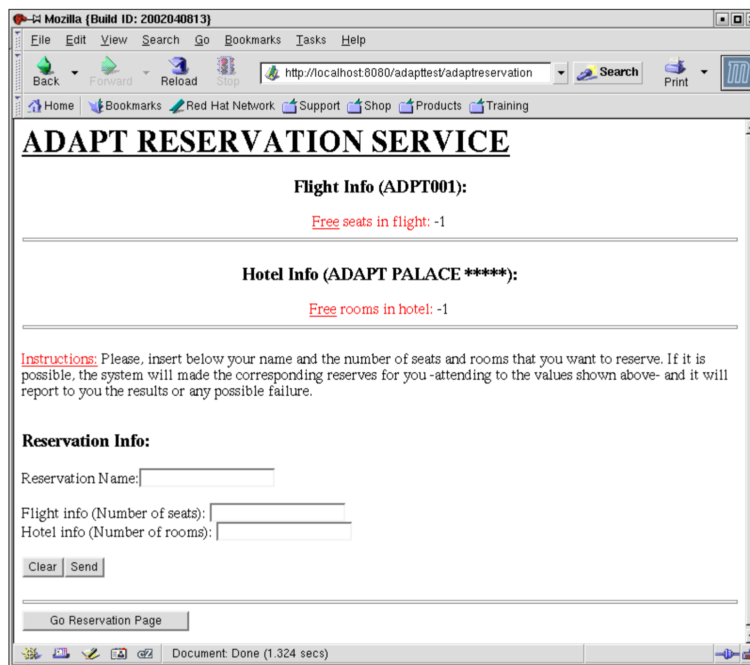


Figure 4: Use of ONT HLS

The application consists of one activity with two nested sequential activities: one for the flight reservation and another for the hotel booking. The top level activity is created when the servlet invokes the `makeReservation` method on the TravelAgency EJB (Listing 2, line 9). The ONT service is contacted in the `getUserOpenNested` private method (Listing 1). The top level activity commits if both the tickets and the hotel rooms can be booked (line 13). Otherwise, the top level activity is rolled back

(lines 17, 20, 23). The invocation of the `reserveSeats` method creates a nested activity (Listing 3, line 4) that checks the availability of seats. If there are enough seats the activity will commit (line 24). The commit method takes as parameter the compensation action (lines 22, 23). The ONT service will invoke the compensation to undo the flight reservation, if the top level activity does not succeed.

Compensation actions are classes that implement the `Compensator` interface (Listing 4). This class must implement a `compensate` method (Listing 4, line 5), which will be invoked by the ONT service to compensate the corresponding activity. The compensate method in this class undoes the ticket reservation invoking the `unreserveSeats` method of the `airline` EJB (Listing 4, line 8). The `unreserveSeats` method is executed as an activity (Listing 5, line 4). This method just returns the tickets, i.e, it increments the number of available seats in the flight (Listing 5, line 14), and commits the activity (Listing 5, line 15). If the database operation fails, the activity rolls back (Listing 5, line 21).

Listing 1: Getting the UserOpenNested interface

```
1  private UserOpenNested getUserOpenNested() {
2     Context ctx = null;
3
4     try {
5        ctx = new InitialContext();
6        uon = (UserOpenNested) ctx.lookup("UserOpenNested");
7     } catch (...) {
8     ...
9     }
10    return uon;
11 }
```

Listing 2: Travel Agency EJB code

```
1  public void makeReservation(String name, int nSeats, int nRooms)
2     throws Exception {
3
4     Airline airline = getAdaptAirlineEJB();
5     Hotel hotel = getAdaptHotelEJB();
6     UserOpenNested uon = getUserOpenNested();
7
8     try {
9        uon.activityBegin(0);
10       try {
11          airline.reserveSeats(nSeats);
12          hotel.reserveRooms(nRooms);
13          uon.activityCommit(null);
14       } catch (RemoteException e) {
15       ...
16       } catch (NotEnoughSeatsException e) {
17          uon.activityRollback();
18          throw e;
19       } catch (NotEnoughRoomsException e) {
20          uon.activityRollback();
21          throw e;
22       } catch (SQLException e) {
```

```
23            uon.activityRollback();
24            throw e;
25         }
26      } catch (...) {
27      ...
28      }
29   }
```

Listing 3: Airline EJB code

```
1  public int reserveSeats(int nSeats) throws NotEnoughSeatsException,
      SQLException {
2     UserOpenNested uon = getUserOpenNested();
3     try {
4        uon.activityBegin(0);
5        Context ctx = new InitialContext();
6        DataSource ds = (DataSource) ctx.lookup("java:/AdaptAirlineDS");
7        Connection c = ds.getConnection();
8        Statement stmt = c.createStatement();
9        ResultSet rs = stmt.executeQuery("SELECT seats FROM flights");
10       int freeSeats = 0;
11       rs.next();
12       freeSeats = rs.getInt(1);
13       if ((freeSeats - nSeats) < 0) {
14          uon.activityRollback();
15          rs.close();
16          stmt.close();
17          c.close();
18          throw new NotEnoughSeatsException();
19       } else {
20          stmt.executeUpdate(
21             "UPDATE flights SET seats = " + (freeSeats - nSeats));
22          ReserveSeatsCompensation compensation =
23             new ReserveSeatsCompensation(this, nSeats);
24          uon.activityCommit(compensation);
25       }
26       rs.close();
27       stmt.close();
28       c.close();
29    } catch (SQLException e) {
30       try {
31          uon.activityRollback();
32       } catch (...) {
33       ...
34       }
35    } catch (NotEnoughSeatsException e) {
36       throw e;
37    }
38    ...
39 }
```

Listing 4: Compensation class for reserveSeats

```
1  public class ReserveSeatsCompensation implements Compensator {
2     AirlineBean airline;
3     int seats;
4     ...
5     public void compensate() {
6        System.out.println("AIRLINE RESERVATION COMPENSATION");
7        try {
8           airline.unreserveSeats(seats);
9        } catch (CompensationErrorException e) {
10          System.out.println(
11          "AIRLINE RESERVATION COMPENSATION NOT PERFORMED DUE TO A DB ERROR
               !!!");
12       }
13       System.out.println("AIRLINE RESERVATION COMPENSATION SUCCEEDED!!!");
14    }
15    ...
16 }
```

Listing 5: unreserveSeats method of the Airline EJB

```
1  public int unreserveSeats(int nSeats) throws SQLException {
2     UserOpenNested uon = getUserOpenNested();
3     try {
4        uon.activityBegin(0);
5        Context ctx = new InitialContext();
6        DataSource ds = (DataSource) ctx.lookup("java:/AdaptAirlineDS");
7        Connection c = ds.getConnection();
8        Statement stmt = c.createStatement();
9        ResultSet rs = stmt.executeQuery("SELECT seats FROM flights");
10       int freeSeats = 0;
11       rs.next();
12       freeSeats = rs.getInt(1);
13       stmt.executeUpdate(
14          "UPDATE flights SET seats = " + (freeSeats + nSeats));
15       uon.activityCommit(null);
16       rs.close();
17       stmt.close();
18       c.close();
19    } catch (SQLException e) {
20       try {
21          uon.activityRollback();
22       } catch (...) {
23       ...
24       }
25       throw new CompensationErrorException();
26    } catch (...) {
27    ...
28    }
29    return 0;
30 }
```

5 ONT HLS Implementation

The ONT HLS implements the specific signalset, signals, actions and outcomes required by the open nested transaction model.

Applications use the `UserOpenNested` interface, which provides the `activityBegin`, `activityCommit` and `activityRollback` methods.

The `activityBegin` method invokes the AS to create an activity and invokes the JTA to create a transaction. Any existing transaction on the current thread is suspended before the creation of the new activity and transaction.

The `activityCommit` invokes the JTA to commit the associated transaction. If the transaction commits and a `Compensator` object was provided in the `activityCommit`, and the commit corresponds to a nested activity, the ONT HLS adds an `action`, responsible for compensation, to the parent activity. This action specifies interest in the ONT signalset. If the commit is related to a top-level activity, the AS is invoked to complete the activity with `CompletionStatusSuccess` using the ONT signalset. If the transaction aborts, the AS is invoked to complete the activity with status `CompletionStatusFail`.

The `activityRollback` method invokes the JTA to roll back the associated transaction and then invokes the AS to complete the activity with status `CompletionStatusFail`.

The ONT service creates two signals: `activity_rolledback` and `activity_committed`, and the following outcomes: `parent_add_successful`, `parent_has_completed`, `parent_add_failed`, `compensate_successful`, and `compensate_failed`. The signalset is invoked by the AS once an activity completes. The AS provides the signalset with the completion status of the activity. The signalset returns the `activity_committed` signal if the completion status was `CompletionStatusSuccess`, or the `activity_rolledback` signal if the completion status was `CompletionStatusFail`.

In the ONT model, compensation objects are wrapped by actions. If an action receives the `activity_rolledback` signal, it will invoke the compensation. If an action receives the `activity_committed` signal, the action will register the compensation object with the parent activity coordinator, if any. If there is no parent, the compensation object is not used and the action returns `compensate_successful`. If the compensation object is registered with the parent activity, the action returns the outcome `parent_add_successful`. If the action fails to register the compensation object, the action returns the outcome `parent_has_completed` if the parent has completed, or `parent_add_failed` if the registration with the parent fails due to any other reason.

If a parent activity fails (`CompletionStatusFail`) or any ancestor, its committed children (descendants) will be compensated. Compensation happens by invoking the corresponding compensation objects when the signal `activity_rolledback` is sent. If the compensation succeeds, the action returns the outcome `compensate_successful`. Otherwise, it returns the outcome `compensate_failed`.

5.1 Deploying the Activity Service and an HLS

In order to be available to the HLS, the AS must be registered as a service into a registration service (JNDI). The AS uses JNDI to publish its interfaces (e.g. java:comp/UserActivity and services:activity/ActivityManager).

An HLS implementation may also require a registration service to register its public interface and a transaction service, if it needs to demarcate transactions.

We have integrated and tested our implementation with the JBOSS application server (vesion 3.2.1). Both the AS and the ONT HLS register their interfaces using the JBOSS JNDI registration service. The ONT HLS uses the JBOSS transaction service in order to demarcate transactions.

6 Integration with Web Services

This deliverable is aimed to provide the core support for advanced transaction models. Although, the integration with web services is not within the deliverable scope, we include a couple of use cases on this topic to illustrate how this integration can be materialized.

There are two ways of using the activity service within a web service:

1. **A non-conversational web service.** An operation of a web service can begin one or more activities, but the operation completes all the activities before returning control to the client. In this case there is nothing specific on the use of an HLS. The implementation of the web service operation will invoke an HLS as in the travel agency example.
2. **A conversational web service**. The behaviour of a conversational web service is similar to the one of statefull session beans in J2EE. A conversational web service may keep information (activities) about previous invocations from a client. In a conversational web service an activity spans several invocations of a web service from the same client, i.e., an operation of a web service begins one or more activities, but does not complete all of them. Other web service invocations from the same client can be executed within the scope of activities started by previous web service invocations. The activity service specification mandates that activities are associated to threads. However, each web service invocation can be executed by different threads in the SOAP engine. This raises the problem that ulterior invocations within the same conversation would not be associated with the activities started by the previous invocations from that client. For this reason, the ONT HLS has been extended with operations to suspend and resume ONT activities across invocations. Before completing a web service invocation part of a conversation, it is necessary to suspend the current activity and store it as part of a session object keeping the state of the conversation. Ulterior invocations on behalf of the same client will recover the activity and resume it before performing the processing associated to the web service operation.

6.1 A Conversational Web Service

This example extends our previous example of the travel agency into a conversational web service. We have used AXIS 1.1 as a SOAP engine and deployed the web service using a `session` scope.

The travel agency web service contains two operations, which reserve tickets for a flight or a number of rooms in a hotel. Listings 6 and 7 show the java code for the methods that implement these two operations. A client must invoke the operation to reserve tickets (`reserveSeats`) and then the operation to book hotel rooms (`reserveRooms`) in that order. These two operations are executed as an ONT that starts with the `reserveSeats` invocation (Listing 6, line 9) and finishes in the `reserveSooms` operation (Listing 7, lines 11, 14, 21). In order to keep the same ONT across invocations, the activity must be suspended before a web service operation finishes and stored (Listing 6, lines 28, 29). The next operation executed as part of the same conversation will start resuming that activity (Listing 7, lines 6, 7). If a conversation spans more web service operations, each operation will include the code to resume the current activity and the code to suspend. The last operation of the conversation will finish the activity (either committing or rolling it back).

Listing 6: ReserveSeats web service operation

```
1  SimpleSession session = new SimpleSession();
2  ...
3  public int reserveSeats(int in0)
4      throws java.rmi.RemoteException,
5      NotEnoughSeatsException,
6      DBException {
7
8      try {
9          uon.activityBegin(0);
10         airline.reserveSeats(in0);
11     } catch (NotEnoughSeatsException e) {
12         try {
13             uon.activityRollback();
14         } catch (...) {
15         ...
```

```
16          }
17          throw new NotEnoughSeatsException();
18      } catch (SQLException e) {
19          try {
20              uon.activityRollback();
21          } catch (...) {
22          ...
23          }
24          throw new DBException();
25      } catch (...) {
26      ...
27      }
28      ONTActivity activity = uon.suspend();
29      session.set("ACTIVITY", activity);
30      return in0;
31  }
```

Listing 7: ReserveRooms web service operation

```
1   public int reserveRooms(int in0)
2       throws java.rmi.RemoteException,
3       NotEnoughRoomsException,
4       DBException {
5
6       ONTActivity activity = (ONTActivity) s.get("ACTIVITY");
7       uon.resume(activity);
8
9       try {
10          hotel.reserveRooms(in0);
11          uon.activityCommit(null);
12      } catch (NotEnoughRoomsException e) {
13          try {
14              uon.activityRollback();
15          } catch (...) {
16          ...
17          }
18          throw new NotEnoughRoomsException();
19      } catch (SQLException e) {
20          try {
21              uon.activityRollback();
22          } catch (...) {
23          ...
24          }
25          throw new DBException();
26      } catch (...) {
27      ...
28      }
29      return in0;
30  }
```

A Software Distribution

The software distribution of this deliverable consists of:

- A zip file, `jass1-0.zip`, with the installation instructions, class files, and an Ant build file to deploy the activity service, the ONT HLS, and the travel agency example. This file can be downloaded from: `http://adapt.ls.fi.upm.es/private/software/jass1-0.zip`
- A zip file, `jass1-0-src.zip`, with the java source files plus the JavaDoc documentation. This file can be downloaded from: `http://adapt.ls.fi.upm.es/private/software/jass1-0-src.zip`

References

[HLR+03] I. Houston, M. C. Little, I. Robinson, S. K. Shrivastava, and S. M. Wheater. The CORBA Activity Service Framework for Supporting Extended Transactions. *Software Practice and Experience*, 33(4):351–373, 2003.

[JP03] R. Jiménez-Peris and M. Patiño-Martínez. ADAPT Project. Deliverable D5: Transaction Support. *http://adapt.ls.fi.upm.es/deliverables/transactions.pdf*, 2003.

[OMG02] OMG. *Additional Structuring Mechanisms for the OTS Specification 1.0*. September 2002.

[Sun] Sun. *Activity Service and Open Nested APIs*. http://jcp.org/aboutJava/communityprocess/review/jsr095/index.html.

[Sun99a] Sun. *Java Transaction API Specification (JTA) 1.01*. April 1999.

[Sun99b] Sun. *Java Transaction Service (JTS) 1.0*. December 1999.

[Sun03] Sun. *J2EE Activity Service Specification Draft 0.1*. June 2003.

[WS92] G. Weikum and H. J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In A. K. Elmagarmid, editor, *Database Transaction Models*, chapter 13, pages 515–554. 1992.