Java™2 Enterprise Edition

J2EE™ Activity Service Specification

JSR095

Specification Lead:
Ian Robinson
IBM Corp.

Technical comments:
jsr95@hursley.ibm.com

Proposed Community Review Draft 2
Expert Group Draft

Trademarks

1.0 Introduction

As the J2EE environment matures, increasingly complex business applications are placing greater demands on the container/server middleware to support more sophisticated transactional semantics than the short-lived ACID transactions provided by the Java Transaction Service (JTS)[1]. In particular, the web services model requires application designers to consider the correlation of requests over extended periods of time and the difficulties associated with coordinating multiple resources over such periods. Many stategies are available for dealing with extended transactions, some appropriate for one type of application and some appropriate for another. But there is no single extended transaction model that will satisy all types of application; what is required is a middleware framework that can be exploited by arbitrary, specific extended transaction models. The OMG Activity service[2]- specifies such a framework for CORBA-based middleware. This document describes the system design and interfaces for a J2EE Activity service that is the realization, within the J2EE programming model, of the OMG Activity service.

The purpose of the Activity service is to provide a middleware framework on which extended Unit of Work (UOW) models can be constructed. An extended UOW model might simply provide a means for grouping a related set of tasks that have no transactional properties or it may provide services for a long-running business activity that consists of a number of short-duration ACID transactions. The Activity service is deliberately non-prescriptive in the types of UOW models it supports. The advantage of structuring business processes as activities with looser semantics than ACID transactions, for example by modeling a business process as a series of short-duration ACID transactions within a longer-lived activity, is that the business process may acquire and hold resource locks only for the duration of the ACID transaction rather than the entire duration of the long-running activity. In a widely distributed business process, perhaps involving web-based user interactions and cross-enterprise boundaries, it is neither practical nor scalable to hold resource locks for extended periods of time. A typical problem with extended UOW models is that the failure scenarios may be quite complex, potentially involving the compensation of some or all of the ACID transactions that were committed before a long-running activity failed. The responsibility for providing the appropriate recovery from such a failure may be shared between the application itself, which is the component that understands *what* needs to be compensated, and the extended unit of work service provider, which might provide facilities to register compensating actions.

The Activity service provides a generic middleware framework on which many types of extended transaction and other unit of work, models can be built.

1. Java Transaction Service, V1.0, *Sun Microsystems Inc.*
2. Additional Structuring Mechanisms for the OTS Specification - *OMG document orbos/2001-11-08 (http://www.omg.org/cgi-bin/doc?orbos/2001-11-08.).*

1.1 Scope

This document and related javadoc describes the architecture of the J2EE Activity service and defines the function and interfaces that must be provided by an implementation of the J2EE Activity service in order to support high-level services constructed on top of this. Such high-level services provide the specific extended transaction model behavior required by the application component.

Specific high-level services and extended transaction models that use the Activity service are beyond the scope of this specification and should be introduced into J2EE via separate JSRs.

This specification concerns itself with the rendering of the OMG Activity service[2] into the J2EE architecture. Interoperability of Activities distributed across heterogenous implementation domains is ensured by requiring the construction of interoperable Activity service contexts, defined in the org.omg.CosActivity package and described in detail in [2]. Specific requirements for interoperability are described in this specification in "Interoperability" on page 43. The basis for such interoperability is an Activity service context defined in IDL which is wholly appropriate for propagation over IIOP. It is envisioned that alternative schema (for example, XML-based) and bindings will be defined for Activity service context appropriate for its propagation over protocols other than IIOP (for example SOAP/HTTP), and work is in progress to this end, but this is beyond the scope of this specification.

Note that the term *extended transaction model* does not necessarily imply the involvement of any ACID transactions, although it may. Throughout the remainder of this specification, the term *transaction*, if unqualified, will be used to refer to a JTS[1] transaction which is typically accessed via JTA[3] in J2EE.

1.2 Target Audience

The target audience of this specification includes:

- providers of high-level services that offer extended transaction behavior.
- implementors of application servers and EJB containers.
- implementors of transaction managers, such as a JTS.

3. Java Transaction API, V1.0.1, *Sun Microsystems Inc.*

1.3 Organization

This document describes the architecture of the Activity service as it relates to the J2EE server environment. The different roles of the components of the service are described, particularly with respect to higher-level services that are built on top of the Activity service. Specific Activity service interfaces are described in general terms in this document and in more detail in the accompanying javadoc packages.

1.4 Document Convention

A regular Times New Roman font is used for describing the Activity service architecture.

A regular Courier font is used when referencing Java interfaces and methods on those interfaces.

1.5 J2EE Activity Service Expert Group

The J2EE Activity service expert group includes the following:

Ian Robinson, IBM
Alex Mulholland, IBM
Tony Storey, IBM
Tom Freund, IBM
Orest Halustchak, Autodesk Inc.
Ram Jeyaraman, Sun Microsystems Inc.
Mark Little, Hewlett-Packard
Ramesh Loganathan, Pramati Technologies
Eric Newcomer, IONA Technologies
Pyounguk Cho, IONA Technologies
Phil Quinlan, Bank of America
Satish Viswanathan, iPlanet
Martin West, Spiritsoft
Mehmet C. Eliyesil, SilverStream Software Inc.
Roman Kishchenko, SilverStream Software Inc.
Alex Boisvert , Intalio Inc.
Riad Mohammed , Intalio Inc.

1.6 Acknowledgements

In addition to the members of the expert group, Neil Mallam and Thomas Mikalsen, of IBM, have provided invaluable contributions to this specification.

2.0 Overview

A long-running business transaction may be represented as an application activity, A_0, which is split into many different, coordinated, short-duration activities. This is illustrated below in Figure 1. A_1 and A_2 are Activities (represented by broken ellipses) containing JTA transactions (represented by solid ellipses) T_1 and T_2; A_3 and A_4 do not use JTA transactions at all. In this example A_2 and A_3 are executed concurrently after A_1.

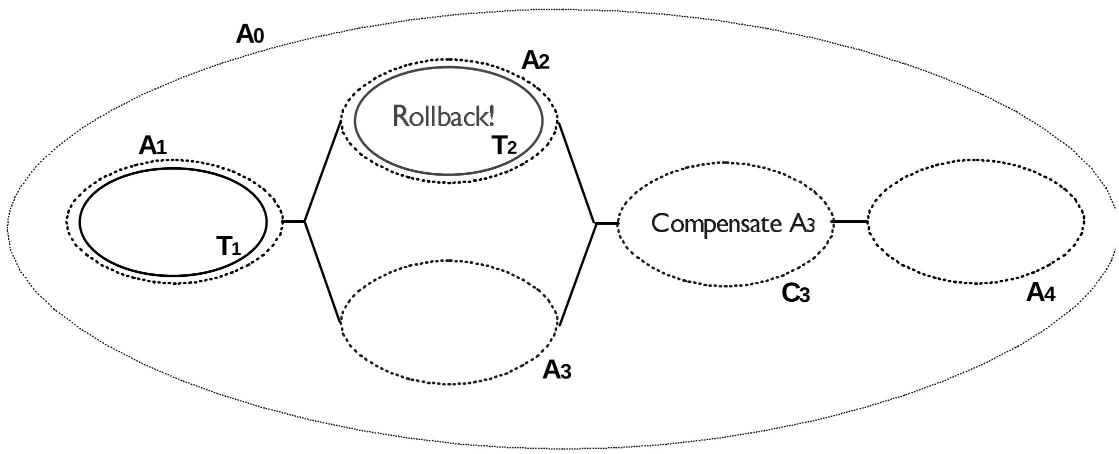

FIGURE 1 A long-running application activity

The reason for structuring the application activity as a *logical long-running transaction* rather than as a single top-level transactions is to prevent certain acquired resources from being held for the entire duration of the application. It is assumed that the application's implementors have segmented the transactional activities within the application into smaller transactional and non-transactional activities, each transaction being responsible for acquiring (and releasing) only those resources it requires. However, if failures and concurrent access occur during the lifetime of these activities then the behavior of the entire *logical long-running business transaction* may not possess ACID properties. Therefore, some form of (application specific) compensation may be required to attempt to return the state of the system to (application specific) consistency.

2.1 Scenarios

A travel agency implements a booking system as a long running application activity. The travel agency provides booking for air, rail, bus, rental car, hotel, etc.

2.1.1 Travel Scenario 1

In this scenario, a client wants to book travel from Boston to Cambridge, England. This example is a bit contrived.

In the first step, the client wants to book a flight from Boston to London Heathrow and then rail from Heathrow to Cambridge. The client is presented with options for departing Boston on July 19, arriving at Heathrow July 20. Rail options are provided for departing July 20 and arriving later that day. The travel agent needs to use two independent systems for booking these two, the airline reservation system for the flight and the separate rail system for the rail trip.

The client chooses options for the airline and rail. The agent's system then starts two parallel transactions, one with the rail system and the other with the airline reservation system. In this particular case, the rail reservation system completes and is committed within that system. However, the airline reservation fails because seats cannot be found for the price that the user wanted and the airline transaction is rolled back.

The client then proceeds to pick a different flight that goes to London Gatwick instead. From that location, the client decides to take a bus instead of train.

The first airline reservation transaction is rolled back by the airline reservation system. For the rail reservation, the travel agent applies a compensating transaction that cancels the rail reservation. At that point, the agent's system submits two new parallel transactions, one to book the new airline reservation and the other to book the bus tickets.

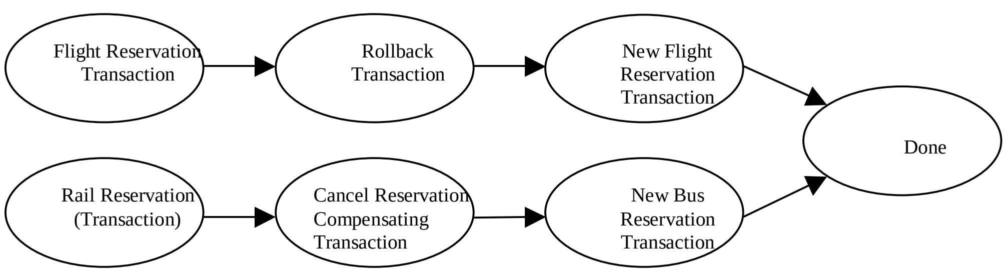

FIGURE 2 Travel reservation scenario

This above scenario could be implemented using ACID transactions between the two systems, but we could say in this scenario that the rail system does not allow a foreign system to hold on to its resource locks.

2.1.2 Travel Scenario 2

This scenario is similar to the above, but includes also hotel reservations and some additional complexity.

In the first step, the client wants to book a flight from Boston to London Heathrow and then rail to Cambridge, as above. The client is presented with options departing July 19 and returning July 29. The client goes ahead and books those reservations. The agent's system executes two parallel transactions against the two independent reservation systems. Both are successful and commit. The client then investigates hotel bookings for Cambridge for the period of stay. The client determines that there are no rooms available during that week for an acceptable price, but that good rooms at a good price are available for the following week.

The client then asks for the rooms to be reserved and then compensating transactions to alter the other bookings for the new travel dates.

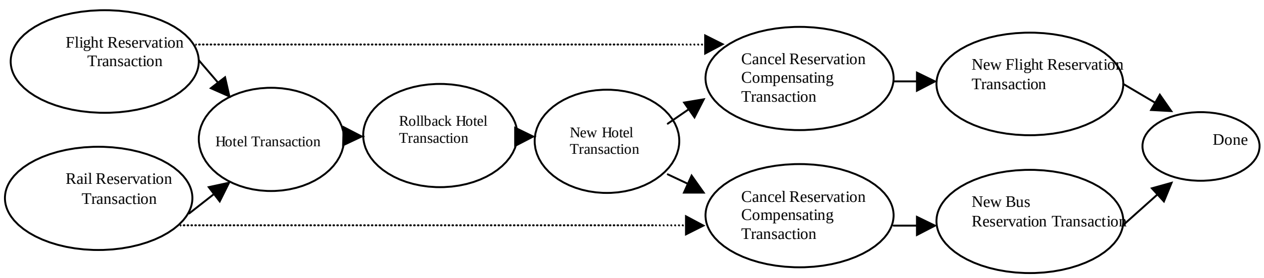

FIGURE 3 Extended Travel Reservation Scenario

This example shows compensating transactions that occur after an intermediate transaction commits (the hotel reservation).

2.1.3 Engineering Design Example

Collaborative design can involve designers from different disciplines working together on a common design, but each responsible for a different aspect of the design.

This engineering design scenario involves designing a new suburban subdivision in a city. It includes designing land parcels, roads, water/sewer services, and electrical services. This city uses a centralized database system that includes the current infrastructure for all land parcels, roads, and facilities. The single database includes data about the current state of land and facilities in the city as well as all design data. This is accomplished by treating the design changes as long running units of work and implementing the High Level Service using a versioning mechanism (similar to Appendix A). The current state is represented with an "as-built" version of the data and each design activity is captured within its own version space. This example shows a "long running" transaction where "long" really is a long period of time – could be months.

First, the project manager begins the long running unit of work in the database as a new project and assigns the design of the land parcels to the land department. The land department designers create the first transaction for the design. A single designer implements the design using a land design product. That design is then inserted into the centralized database as a transaction that captures that change in a new “land design” version space. Then the project manager assigns design of water/sewer to an engineer in the water department and the electrical to an electrical engineer. The water/sewer designer uses a CAD system that runs directly against the database. Their design can view the land design and adds the water/sewer design to the database in a water/sewer version space. Similarly, the electrical engineer adds their design.

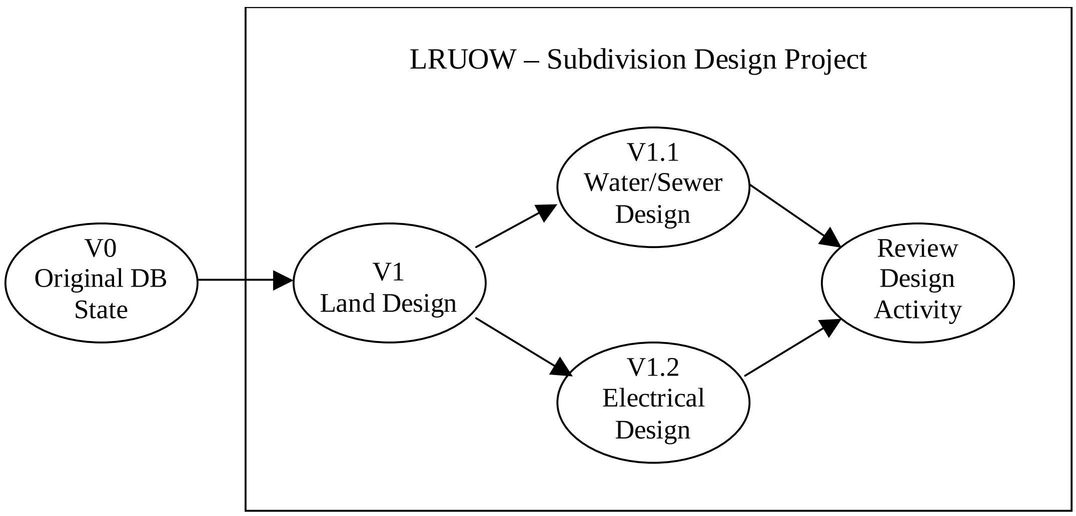

FIGURE 4 **Long-Running Unit of Work Scenario**

At this point, a separate city engineer reviews the design and determines that a change is needed to the land design. The reviewers were very busy and didn’t get around to reviewing the design until a month later at which time the other engineers had produced their designs. At this point the water/sewer and electrical designs need to be removed and a compensating transaction applied to the land design. Then the water/sewer and electrical engineers redo their designs. This is accomplished by first applying compensating transactions against the water/sewer and electrical work by removing those versions of data.

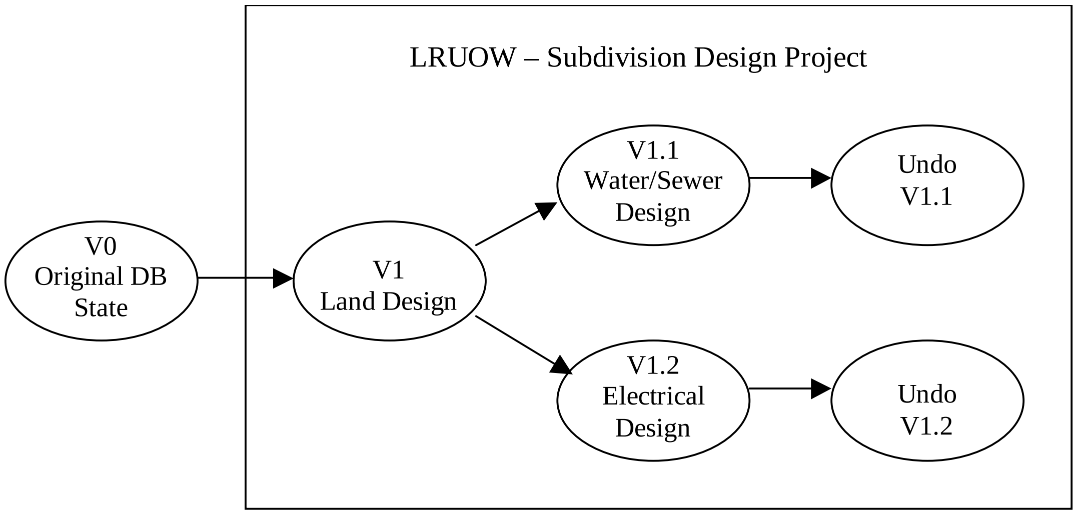

FIGURE 5 **LRUOW Scenario - Undo Intermediate Versions**

Then, a new transaction makes the change to the land design, which is reviewed immediately.

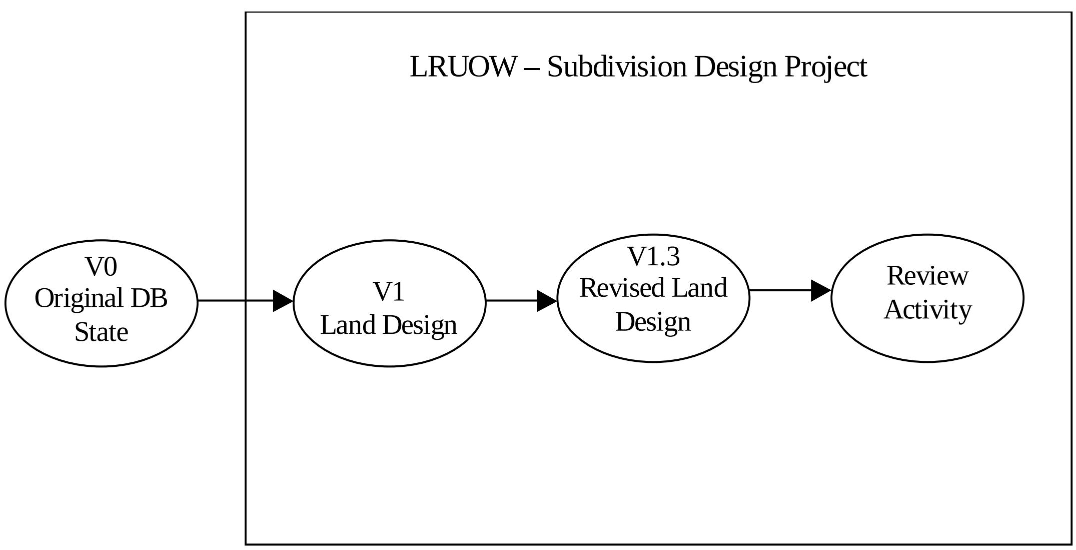

FIGURE 6 **LRUOW Scenario - Create New Intermediate Version**

Next, the water/sewer and electrical designers redo their design work. Then, the work is carried out in the field where the new subdivision is built.

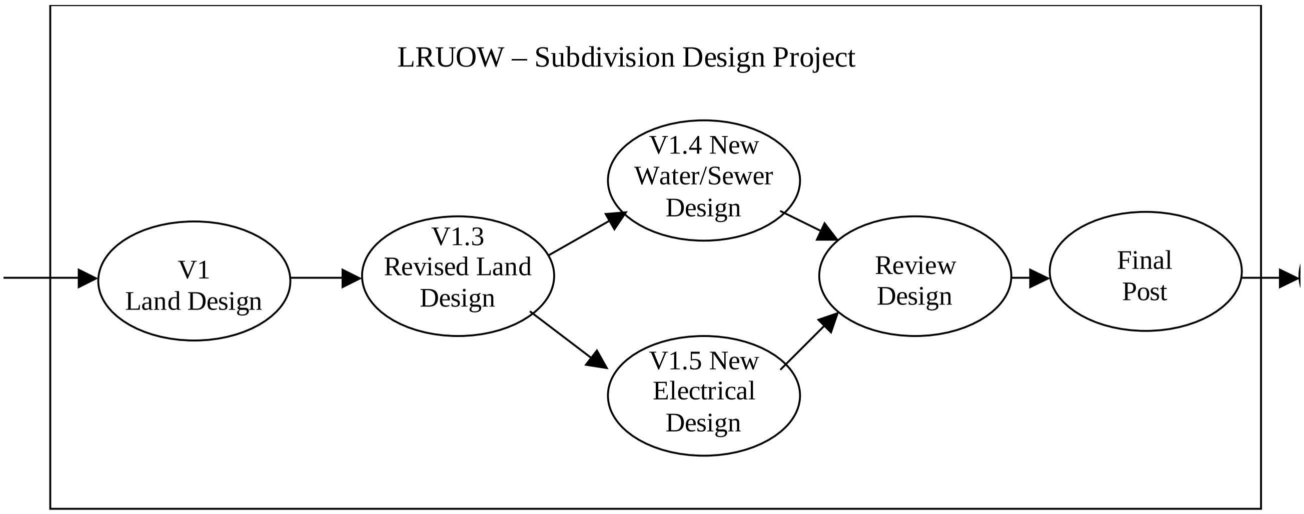

FIGURE 7 LRUOW Scenario - Further Versions are created

Finally, this long running unit of work is done and the versions of data in this long running transaction replace the “as-built” versions in the database using a transaction called “final post”. This basically commits the long running transaction.

This scenario was simplified a fair bit to illustrate the basic ideas.

2.2 The Need for a Framework

There are several ways in which some or all of the application requirements outlined above could be met. However, it is unrealistic to believe that a single high-level model approach to extended transactions is likely to be sufficient for all (or even the majority of) applications. Therefore, the Activity service provides a low-level infrastructure to support the coordination and control of abstract Activities that are given concrete meaning by the high-level services that are implemented on top of the Activity service. These Activities may be transactional, they may use weaker forms of serializability, or they may not be transactional at all; the important point is that the Activity service itself is only concerned with their control and coordination, leaving the semantics of such Activities to the high-level services.

Examples of the types of unit-of-work (UOW) models that may be provided by high-level services that plug into the J2EE Activity service framework include, but are not restricted to, long-running business processes that are transactional only during a final reconciliation phase, sagas with compensation, open nested transactions, workflows, strict two-phase OTS and nested transactions. Specific examples of high-level services that exploit the Activity service architecture are described in Appenix A, “Specific HLS examples” on page 47, and in the OMG Activity service specification[1]. The qualities of service offered by a specific UOW model, and any application architecture considerations that are implied by that model, are factored in the external description of the high-level service and, as such, are beyond the scope of this specification.

2.3 Components of an Activity

An Activity is a unit of (distributed) work that may or may not be transactional. During its lifetime an Activity may have transactional and non-transactional periods. Every entity including other Activities can be part of an Activity, although an Activity need not be composed of other activities. An Activity is characterized by an application-demarcated context under which a distributed application executes. This context is implicitly propagated with all requests made in the scope of the Activity and defines the unit of work scope under which any part of an application executes.

An Activity is created, made to run, and then completed to produce an `Outcome`. Demarcation notifications of any kind are communicated to any registered Activity participants (`Actions`) through `Signals` which are produced by `SignalSets`. A specific UOW model defines the set of `Signals` that may be produced during the lifetime of an Activity and the set of `Outcomes` that result. It also defines a discrete set of state transitions that may occur as the `Signals` are consumed by the Activity participants. These state transitions are encapsulated and managed by the `SignalSet`. `Actions` allow an Activity to be independent of the specific work it is required to do in response to broadcasting a `Signal`. For example, if a JTS were to be implemented as a high-level service (HLS) on top of the Activity service, the `org.omg.CosTransactions.Resources` would be registered as `Actions` with an interest in a two-phase-commit `SignalSet` which produced *prepare*, *commit*, *rollback*, *commit_one_phase* and *forget* `Signals`.

The purpose of the J2EE Activity service specification is to define the roles and responsibilities of the components of such a service implementation in a J2EE server environment and, where appropriate, the J2EE client environment. In particular this specification defines the interfaces and behavior of an Activity service such that vendors may implement high-level services that use these interfaces to provide the desired extended transaction, or other unit of work, models. The details of specific high-level service behaviour and the interfaces between such services and the business applications that use them are beyond the scope of this specification.

3.0 J2EE Activity Service Architecture

The architecture for a high-level service (HLS) providing an extended UOW model and using the facilities of the J2EE Activity service is shown in Figure 8.

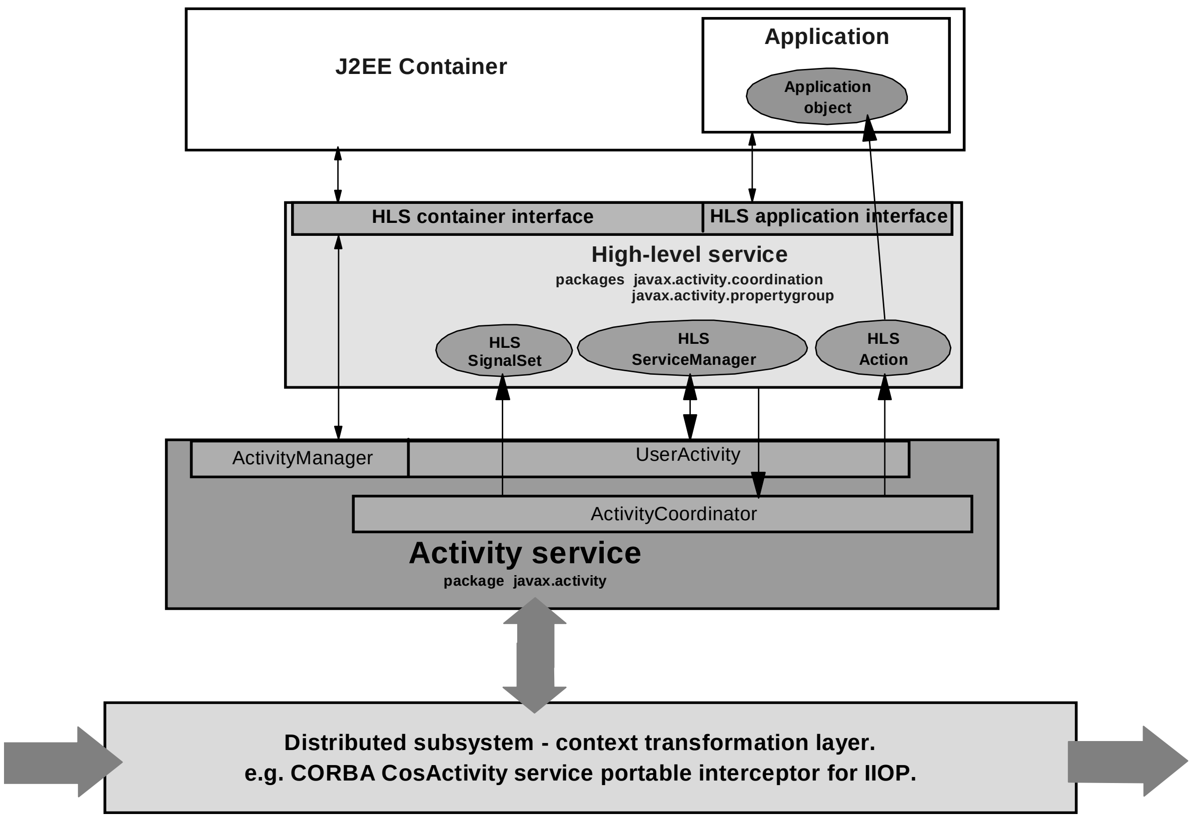

FIGURE 8 **Activity and high-level service architecture**

The architecture is partitioned into the following main components:

Application and container -- the application is designed to participate in a specific type of extended UOW model and uses, either directly or through the container, the facilities provided by the HLS to control the units of activity supported by the HLS. If the HLS provided a compensating extended transaction model, for example, in which a long-running transaction is composed of a sequence of ACID transactions that may need to be compensated following a failure, then the application component would be expected to provide the compensation data that the HLS would drive at the appropriate time. The application component does not call the Activity service directly, but interacts with the Activity service through the HLS. The application component is external to the Activity service and is beyond the scope of this specification.

High-level service (HLS) -- the HLS defines the behavior of a specific extended UOW model and offers interfaces to the application that uses it, as well as the application server and container. The HLS delegates to the Activity service to manage its distributed context and relationships between this context and any JTS context. It uses the Activity service as the means by which signals pertaining to the HLS are distributed to participants in an HLS unit of work. In particular, the HLS provides implementations of the `javax.activity.coordination` interfaces and optionally the `javax.activity.propertygroup` interfaces.
This component is external to the Activity service and is beyond the scope of this specification.

Activity service -- the Activity service manages the HLS's service context, both with respect to other Activity contexts and with respect to JTS context, ensuring its appropriate implicit propagation with remote requests. It provides interfaces to a HLS that support context demarcation and pluggable coordination of HLS-specific objects. The Activity service provides implementations of the classes and interfaces of the `javax.activity` package. This specification is primarily concerned with this component.

Distributed subsystem -- Activity servce context may be distributed across execution domains and potentially between different J2EE providers, and indeed application server architectures. The means by which that context is distributed is dependent on the coupling between the domains. For example, the OMG Activity service defines the interoperable distribution of Activity service contexts over IIOP; the context transformation for distribution over IIOP is provided by an OMG Activity service portable interceptor[4].

This component division is intended to be illustrative rather than prescriptive. For example, a container may provide the function of a high-level service.

4. Interceptors Published Draft with CORBA 2.4+ Core Chapters - *OMG document ptc/2001-03-04*

4.0 Elements of the Activity service

4.1 Features

The features provided by the J2EE Activity service to support the implementation of extended transaction models as high-level activity services are described in this section.

4.1.1 Generic coordination

The Activity service provides a framework for sending generic `Signals` to `Actions`, where both `Signals` and `Actions` are given meaning by and implemented by the high-level service (HLS) that uses the Activity service. The HLS provides a `SignalSet` object that is responsible for producing the `Signals`; the `SignalSet` is obtained from a `ServiceManager`, supplied by the HLS, and is *plugged into* the Activity service `ActivityCoordinator` which drives the `SignalSet`, at the appropriate time, to produce `Signals` and distributes the `Signals` to these `Actions` that have registered an interest. The `ActivityCoordinator` places no semantic meaning on the `Signals` but manages the relationship with registered `Actions` and returns the `Outcomes` received from those `Actions` to the `SignalSet`. The `SignalSet` is a finite state machine that produces `Signals` based on the managed state and accepts `Outcomes` to those `Signals` to influence state transitions. The specific semantics of a HLS are then encapsulated by the `SignalSet` and `Actions` provided by the HLS and the `Signals` and `Outcomes` used by the HLS objects, while the generic management and distribution of the resources of the HLS are provided by the Activity service.

Figure 9 shows the relationship between the `ServiceManager`, `SignalSet`, and `Actions` provided by the HLS, the `Signals` and `Outcomes` used by the HLS objects and the `ActivityCoordinator`.

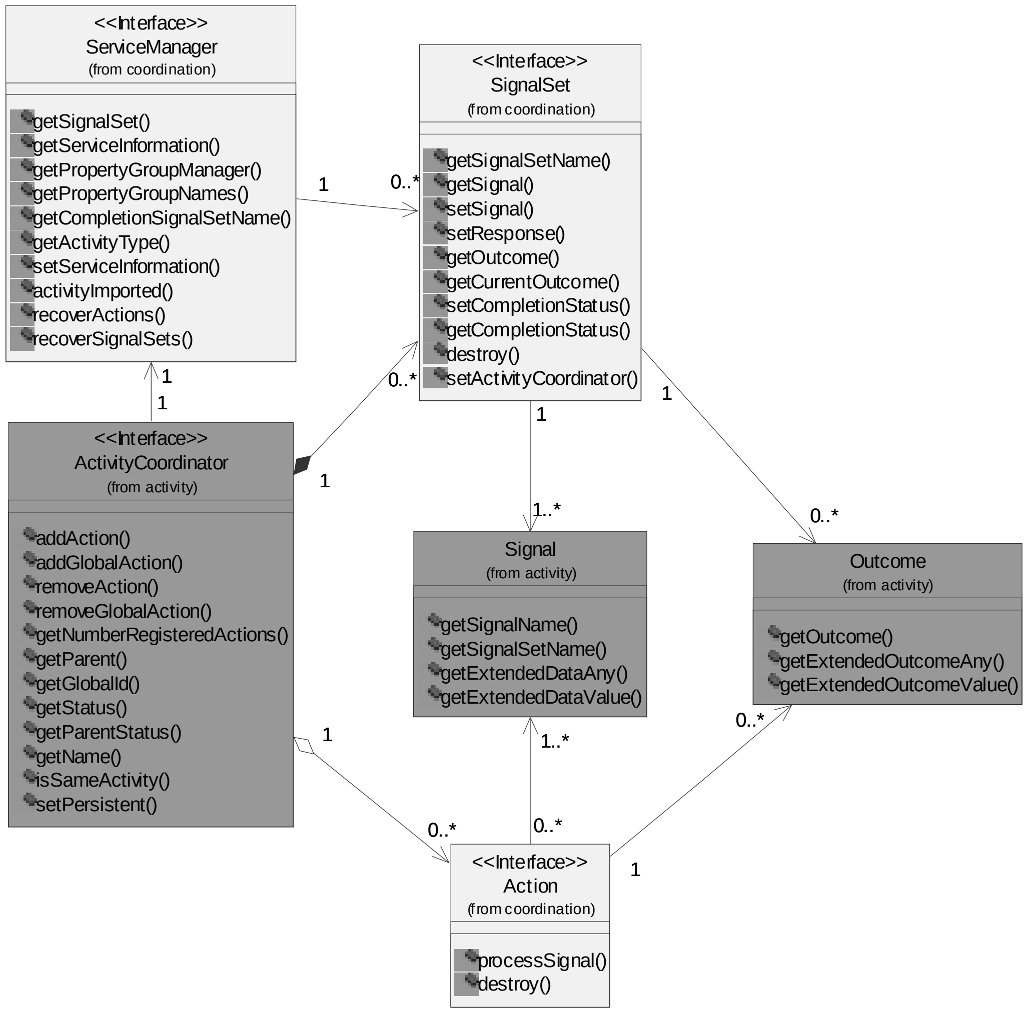

FIGURE 9 **Generic coordination using a pluggable SignalSet provided by the ServiceManager**

4.1.2 Grouping and management of context

Each Activity started by a HLS may be a child of an Activity already running on the thread; such child activities are wholly encapsulated by their parent Activity. An Activity may encapsulate a JTA transaction and may be encapsulated by a JTA transaction. Context hierarchies of Activity and transaction contexts may be created in one execution environment (application server) and propagated via Activity service context to another. The Activity service provides `UserActivity` and `ActivityManager` interfaces to enable a HLS and an EJB container to manage these context hierarchies; the `UserAc-`

`tivity` interface provides simple demarcation methods to begin and complete Activities while the `ActivityManager` interface provides more complex context management functions, such as `suspend` and `resume`. The Activity service hides the complexity of the context hierarchies by enabling the caller of these methods to operate only on the most recent context (the *active* or *current* context) belonging to a particular HLS.

Activities started by different HLS's on the same thread may be wholly unrelated, in which case it would be inappropriate for their contexts to form parent-child relationships and to be cooperatively managed. On the other hand distinct, specific HLS's may wish to have their Activity contexts cooperatively managed with one or more other distinct, specific HLS's. *Context groups* are supported by the Activity service for this purpose such that a HLS may specify, through its ServiceManager, which *context group* its Activities participate in.

All Activity contexts within a particular *context group* are strictly nested with respect to one another but are independent of Activity contexts in any other *context group*. A *context group* is identified by a `String` name; an HLS that wishes to have its contexts managed independently of any other HLS can specify the package name of the HLS as the name of the *context group* it wishes to participate in.

A *default* context group is provided, identified by an empty `String`, which must be used by any HLS that wishes to have its Activity contexts managed cooperatively with JTA context. This default context group provides nesting behavior compatible with that described in the OMG Activity service.

Consider, as an example, two high level services, HLS_a and HLS_b. HLS_a offers a unit of work (UOW) scope that consists of an Activity that implicitly starts a JTA transaction. HLS_b offers a UOW scope that is used to control execution parameters determine by context specific to the client instance. An application may consist of components that use either or both of the contexts of HLS_a and HLS_b . While HLS_a contexts must be strictly managed with respect to JTA contexts, HLS_b contexts are wholly independent of HLS_a contexts and are not be affected by them.

If the JTA support is provided by a JTS implementation, then it is JTS contexts that are managed cooperatively with Activity contexts. A JTA provider does not have to use a JTS implementation but is required to represent the transaction context as a JTS context for remote interoperation.

4.1.3 Distributed service context

The behavior of distributed components running under an Activity context is made location independent by use of an Activity service context that is propagated implicitly on all remote calls. For remote calls over IIOP an Activity service implementation should provide a portable interceptor[4] to enable context from a client to be implicitly propagated and established in a target server environment. Response context from the server is returned to the client by the same mechanism.

The format of the interoperable service context is defined in the OMG Activity service specification[2].

4.1.4 Distributed application property data

A distributed Activity may consist of a number of components distributed over a variety of remote servers. Different components within the Activity may share application-specific property data that is scoped to the Activity via `PropertyGroup` context that is a subset of the Activity service context, managed by the application and HLS. The relationship in the `PropertyGroup` data between parent and child Activities is defined by the `PropertyGroupManager`, provided by the HLS.

4.2 Activity service packages

The Activity service specification defines three new `javax` packages, described briefly in this section and in further details in the javadoc that accompanies this specification. The `javax.activity` package contains interfaces and classes provided by the Activity service itself. The `javax.activity.coordination` and `javax.activity.propertygroup` packages contain interfaces that may be implemented by a HLS.

4.2.1 javax.activity *package*

The classes and interfaces of the `javax.activity` package are provided by the Activity service itself. These are summarized in this section and described in full in the accompanying javadoc.

4.2.1.1 UserActivity

A `javax.activity.UserActivity` instance is used by each HLS to control demarcation of Activities, through the `begin` and `complete` and `completeWithStatus` methods and to provide access to other Activity service interfaces, such as the `ActivityCoordinator`. An instance of `UserActivity` is obtained, by an HLS, via a JNDI lookup of **`java:comp/UserActivity`**. The HLS must register its `javax.activity.coordination.ServiceManager` implementation with the Activity service, through the `UserActivity.registerService` method, before the `UserActivity` instance may be used to start new Activities. The `ServiceManager` is used by the `UserActivity` to determine specific behavior of Activities it creates, such as the `PropertyGroups` and default completion `SignalSet` they use.

Each Activity started by a `UserActivity` instance logically creates an Activity context to represent a unit of work instance specific to the HLS whose `ServiceManager` is registered with the `UserActivity` instance. Methods of the `UserActivity` interface that operate on the active Activity context are operating on the HLS Activity context most recently associated with the calling thread. In practise, the objects that represent an Activity, such as the ActivityContext and the ActivityCoordinator, may be lazily created as required during the lifetime of the Activity.

4.2.1.2 ActivityManager

A `javax.activity.ActivityManager` instance is used by a HLS or EJB container for advanced context management of Activities, such as `suspend` and `resume`. These operations are typically executed as a result of a container policy defined by a HLS. An HLS is responsible for ensuring, through the specification and implementation of its EJB container policies, that the Activity Service context(s) active at the completion of an EJB method are the same as those that were active prior to the method invocation.

The `ActivityManager` interface is a specialization of `UserActivity`, with which an HLS should register its `ServiceManager`. An instance of `ActivityManager` is obtained via a JNDI lookup of **`services:activity/ActivityManager`**. `ActivityManager` instances are only available in EJB and web container environments, whereas `UserActivity` may be made available through a client container. A container that provide access to the `ActivityManager` may provide a transient binding at the specified **`services:`** URL (which it may re-initialise in-memory each time the container is initialised) but is required to provide access to authorized components, once initialised, regardless of the active J2EE component. For example, a portable interceptor should be able to perform a lookup of this object even in the absence of an active J2EE component.

4.2.1.3 ActivityToken

A `javax.activity.ActivityToken` is used to manipulate hierarchies of Activity and transaction contexts via the `suspend` and `resume` operations of the ActivityManager interface.

ActivityTokens are local to the execution process but may be used on any thread within the execution process.

4.2.1.4 CompletionStatus

The `javax.activity.CompletionStatus` interface defines a finite set of 3 states that an Activity may complete in:

CompletionStatusSuccess -- The Activity has successfully performed its work and can complete accordingly. When in this state, the Activity `CompletionStatus` can be changed.

CompletionStatusFail -- The Activity has not successfully completed its work, either as a result of application failure or simply due to processing that is not yet complete, and should be driven accordingly during completion. When in this state, the Activity `CompletionStatus` can be changed. This is the initial `CompletionStatus` of an Activity.

CompletionStatusFailOnly -- The Activity has not successfully completed its work, as a result of a system or application failure, and should be driven accordingly during completion. When in this state, the Activity `CompletionStatus` cannot be changed.

4.2.1.5 Status

The `javax.activity.Status` interface defines a finite set of states that an Activity may progress through during its lifetime.

StatusActive -- There is an active Activity associated with the calling thread.

StatusCompleting -- The Activity associated with the calling thread is completing.

StatusCompleted -- The Activity associated with the calling thread has completed.

StatusNoActivity -- There is no Activity associated with the calling thread.

StatusUnknown -- The Activity service is unable to determine the status of the Activity associated with the calling thread. This is a transient condition.

StatusError -- The Activity service cannot contact the application's signal set to retrieve signals.

4.2.1.6 GlobalId

The `javax.activity.GlobalId` object uniquely identifies an Activity across the namespace.

This object's `equals` method returns *true* if the parameter object represents the same Activity as the target object.

4.2.1.7 Signal

`Signals` are events that are broadcast to interested parties as part of a coordinated `SignalSet`. Each `javax.activity.Signal` is uniquely identified by a combination of its `SignalName` and the name of the containing `SignalSet`. `Signals` are produced by `javax.activity.coordination.SignalSet` objects and consumed by `javax.activity.coordination.Action` objects.

4.2.1.8 Outcome

A `javax.activity.Outcome` is produced by, and given meaning by, an `Action` which has processed a `Signal`, or by a `SignalSet` when it has finished producing `Signals`. A *completion* `SignalSet` produces such an `Outcome` and this is returned on the `complete` and `completeWithStatus` methods of the `UserActivity` interface.

4.2.1.9 ActivityCoordinator

The javax.activity.ActivityCoordinator is responsible for broadcasting Signals to registered Actions. The ActivityCoordinator obtains the Signals, during broadcasting or completion, from SignalSets provided by the HLS. It has no logic to understand the Signals produced by a SignalSet or the meaning of the Outcomes produced by Actions, it simply mediates between a SignalSet and its registered Actions. The only event-processing logic it possesses is to handle ActionErrorExceptions or unchecked exceptions from Actions by reporting these to the SignalSet through the pre-defined Outcomes with names “ActionError” and “ActionSystemException” respectively. The exception is encoded in the specific data of the Outcome, retrieved through the Outcome object’s getExtendedOutcomeValue method.

There is a single logical ActivityCoordinator instance per Activity, although in an Activity distributed over several application servers there will be an instance of an ActivityCoordinator local to each application server. In such a configuration, the application server on which the Activity is created contains a *root* ActivityCoordinator and each application server to which the Activity context is propagated contains an interposed ActivityCoordinator which is subordinate to the ActivityCoordinator on the server from which the Activity context was propagated. Subordinate ActivityCoordinators register an Action with their superior ActivityCoordinator in order to form a distributed coordination tree. Such an Action is registered with an interest in the superior’s SignalSet(s) for which the local Actions on the subordinate have an interest.

This object’s equals method returns *true* if the parameter object represents the same Activity as the target object.

4.2.1.10 ServiceInformation

An instance of a javax.activity.ServiceInformation object is associated with each Activity and contains information about the HLS to which the Activity belongs. This information is propagated as part of the org.omg.CosActivity.ActivityIdentity structure of the Activity service context, in the activity_specific_data field, when the type field of the ActivityIdentity indicates a J2EE Activity, as described in “Interoperability” on page 43.

4.2.1.11 ActivityInformation

The javax.activity.ActivityInformation class is provided by the Activity service to assist an Action that has registered interest with a system SignalSet to extract the information from Signals produced by that SignalSet. ActivityInformation objects contain information such as the GlobalId of the Activity and are contained in the extended data of system SignalSets.

System SignalSets are described in “Predefined Outcomes and SignalSets” on page 25.

4.2.1.12 PropertyGroupContext

The `javax.activity.PropertyGroupContext` utility object is provided by the Activity service to assist a `javax.activity.propertygroup.PropertyGroupManager` read and write `org.omg.CosActivity.PropertyGroupIdentity` context data during marshaling and unmarshaling of the `org.omg.CosActivity.ActivityContext` that incorporates it. Marshaling and unmarshaling occurs at an execution environment boundary when the context needs to be converted to or from the CDR encapsulated form used for remote propagation over IIOP.

4.2.1.13 CoordinationInformation

A `javax.activity.CoordinationInformation` object is produced by a `SignalSet` and used by the `ActivityCoordinator` during signal-processing to determine how to proceed after a response from a particular `Action` has been processed by the `SignalSet`.

4.2.2 javax.activity.coordination *package*

The interfaces of the `javax.activity.coordination` package are implemented by the HLS that uses the Activity service. These are summarized in this section and described in full in the accompanying javadoc.

4.2.2.1 ServiceManager

A `javax.activity.coordination.ServiceManager` is an entity that is provided by a HLS that uses the Activity service; it is a factory for the HLS's objects, such as the `SignalSets` used by the HLS, and also specifies how the HLS's Activities should be managed. In particular, it is used to specify:

- which `PropertyGroups` the HLS uses
- the default *completion* `SignalSet` that is used to complete the HLS's Activities.
- the `ServiceInformation` for the HLS's Activities (which indicates which ContextGroup the HLS participates in). This is propagated as part of the Activity service context.

The Activity service uses the `ServiceManager` when it creates and operates on Activities specific to that service.

A `ServiceManager` implementation must be bound into JNDI by the HLS provider at a location identified by the *ServiceName* that is returned from `ServiceManager.getServiceInformation().getServiceName()`. The Activity service needs to be able to locate a `ServiceManager` implementation from its *ServiceName* when it imports a service context containing a J2EE Activity.

If an imported `ActivityContext` contains Activities of a `type` other than a J2EE Activity, then an administratively configured URL may be obtained by the Activity service and a `ServiceManager` for non-J2EE Activities could be provided, for example to identify appropriate `PropertyGroupManager`(s) for any received `PropertyGroup` contexts. In the absence of such an administratively-configured ServiceManager, an Activity service implementation that receives non-J2EE Activity contexts may either throw an `InvalidActivityException` or may follow the behaviour specified in “Behaviour in the case of unknown Activity types, ServiceNames or PropertyGroups” on page 45, using the default context group.

4.2.2.2 SignalSet

A `javax.activity.coordination.SignalSet` is an entity that is provided by a HLS built on top of the Activity service that produces `Signals` and understands the responses to those `Signals`. The `SignalSet` abstracts from the `ActivityCoordinator` the knowledge of which `Signal` should be distributed to the registered `Actions` based on the state of the Activity and responses to previous `Signals`. The Activity service itself then needs to provide only a very generic `ActivityCoordinator` to drive any specific `SignalSet`. The `ActivityCoordinator` simply asks a `SignalSet` for the next `Signal` and then broadcasts it to each interested `Action` in turn. The response from each `Action` is fed back to the `SignalSet` which has the knowledge of what that result means, which `Signal` should be sent next and whether the `Action` that returned a particular `Outcome` expects to receive further `Signals`.

If an Activity is distributed between execution domains, then each domain will contain a local `ActivityCoordinator`/`SignalSet` pair with one pair subordinate to the root pair. The root `SignalSet` produces signals whereas the subordinate `SignalSet` merely redistribues received signals to local `Actions` and reconciles the responses. Although the subordinate `SignalSet` has no role to produce signals, it may do so as an optimization if it can predict the next signal from its superior - for example a JTS `SignalSet` can be confident of receiving a *rollback* signal after a *VoteRollback* response.

4.2.2.3 Action

A `javax.activity.coordination.Action` is an entity that is provided by the HLS and registered with an interest in one or more `SignalSets`. An `Action` may only be registered with a single `ActivityCoordinator`.

An `Action` is the target object to which a `Signal`, produced by a `SignalSet`, is sent during the `broadcast`, `complete` and `completeWithStatus` operations initiated via `UserActivity`.

4.2.3 javax.activity.propertygroup *package*

The interfaces of the `javax.activity.propertygroup` package may be provided by an HLS that uses the Activity service, although they are all optional. These are summarized in this section and described in full in the accompanying javadoc.

4.2.3.1 PropertyGroup

A `javax.activity.propertygroup.PropertyGroup` is used to provide distributed context, scoped to an Activity, that may be set by an application or a HLS built on top of the Activity service. The format of the distributed context is specific to the `PropertyGroup` implementation and is neither examined nor understood by the Activity service.

The semantics of the behavioral relationship between `PropertyGroups` in nested Activities is defined by the specification of each type of `PropertyGroup` and not by the Activity service. Any number of named `PropertyGroup` types may be configured in a `ServiceManager` and used within an Activity. When an Activity is started, an instance of each type of `PropertyGroup` used by the Activity is created and associated with the Activity.

4.2.3.2 PropertyGroupManager

A `javax.activity.propertygroup.PropertyGroupManager` is an entity that may be provided by a HLS and understands how to create and manipulate a specific type of `PropertyGroup`. It is registered with the Activity service and is used by the Activity service to create `PropertyGroup` instances and to manipulate the `PropertyGroupContext` that is implicitly propagated as part of an Activity context.

For a particular type of `PropertyGroup`, there must be a `PropertyGroupManager` available (from the `ServiceManager`) in each client and server execution environment for which the `PropertyGroup` will be accessed. If `PropertyGroupContext` is propagated, as part of an Activity context, to an environment in which there is no appropriate `PropertyGroupManager` registered, then the `PropertyGroupContext` is not available within that environment although it may be cached by the Activity service and propagated on to any downstream environment to which the Activity context is further distributed.

4.3 Predefined Outcomes and SignalSets

The Activity service provides implementations of the following predefined `Outcomes` and `SignalSets`.

4.3.1 Outcomes

4.3.1.1 ActionError

This pre-defined `Outcome` is created by the `ActivityCoordinator` and returned to a `SignalSet` if the ActivityCoordinator receives an `ActionErrorException` on an `Action.processSignal()` request.

4.3.1.2 ActionSystemException

This pre-defined `Outcome` is created by the `ActivityCoordinator` and returned to a `SignalSet` if the `ActivityCoordinator` receives a system exception on an `Action.processSignal()` request. The received exception is passed back to the `SignalSet` in the extended data of the `Outcome` and can be retrieved via the getExtendedOutcomeValue method. A SignalSet may handle this in any way it deems appropriate.

4.3.2 SignalSets

4.3.2.1 Synchronization

The *org.omg.CosActivity.Synchronization* `SignalSet` contains the Signals `preCompletion` and `postCompletion`, which are sent to interested `Actions` under the following circumstances:

preCompletion -- sent prior to distributing `Signals` from a completion `SignalSet` if the `CompletionStatus` is *CompletionStatusSuccess*. `Actions` must respond to this signal with a pre-defined Outcome of `preCompletionSuccess` or `preCompletionFailed`.

postCompletion -- sent after all Signals produced by a completion `SignalSet` have been distributed. A `null` Outcome is expected in response to this signal.

No `Outcome` is produced by this `SignalSet`. This `SignalSet` changes the Activity `CompletionStatus` to *CompletionStatusFailOnly* in the event that any `preCompletion` signal receives an Outcome of `preCompletionFailed`, `ActionError` or `ActionSystemException`.

4.3.2.2 ChildLifetime

The *org.omg.CosActivity.ChildLifetime* `SignalSet` contains the signal `childBegin`, which is sent to interested `Actions` under the following circumstances:

childBegin -- sent when a child Activity context is started. This `Signal` is sent after the child Activity and all its `PropertyGroups` have been created, when the child Activity context is the active context on the thread.

No `Outcome` is produced by this `SignalSet`. In the event of any failures being reported back to the `SignalSet` during the processing of the `childBegin` signal, for example as a result of an `ActionErrorException` being raised by any of the registered `Actions`, then the child Activity's `CompletionStatus` is changed to *CompletionStatusFailOnly*. The parent Activity's `CompletionStatus` may or may not be changed as a result of such a failure.

Note to Reviewers: *The OMG Activity service specification has an open Issue (#4711), at the time of writing of this draft of the J2EE Activity service specification, regarding the means by which a* `childBegin` *signal can be distributed in the case where a child Activity is started on a node (JVM) that is not the root node of the parent Activity. A suggested resolution to this issue is that the* `childBegin` *signal be distributed from the node in which the child Activity is started. Actions registered with the parent's upstream superior ActivityCoordinator are not then informed of these events.*

4.3.2.3 Failure

The *org.omg.CosActivity.Failure* `SignalSet` contains the signals `initialFailure` and `finalFailure` , which are sent to `Actions` in the event that a remote `SignalSet`, in which the `Actions` have an interest, cannot be reached during signaling. All `Action` implementations must be prepared to handle these signals and respond appropriately.

initialFailure -- indicates that the application `SignalSet` could not be contacted but that the problem may be transient. An `Action` that receives the `initialFailure` signal should respond with one of two pre-defined Outcomes `org.omg.CosActivity.Failed` or `org.omg.CosActivity.FailureRetry`. Any `Action` that responds with `Failed` will not receive any further signals. Any `Action` that responds with `FailureRetry` is indicating that it wishes the `ActivityCoordinator` to continue to retry contacting the application `SignalSet`. If contact is subsequently made, signaling with the application `SignalSet` may continue.
The Activity service changes the Activity Status to `StatusUnknown` prior to distributing this signal.
The Signal object's getSignalSetName method returns the name of the failed SignalSet rather than "*org.omg.CosActivity.Failure*".

finalFailure -- indicates that the `ActivityCoordinator` has exhausted its attempts to contact the `SignalSet`. The point at which this happens is a detail of the Activity service implementation and may be configured administratively. This signal indicates to the `Action` that it should perform whatever processing is appropriate to it in this situation. The `Failure` SignalSet ignores any `Outcome` produced for this signal.
The Activity service changes the Activity Status to `StatusError` prior to distributing this signal.
The Signal object's getSignalSetName method returns the name of the failed SignalSet rather than "*org.omg.CosActivity.Failure*".

If the application `SignalSet` does not complete its signaling, the `ActivityCoordinator` raises the ActivityNotProcessed exception and this is returned on the UserActivity `complete`, `completeWithStatus` or `broadcast` method that triggered the signaling.

4.4 Recovery

The meaning of a `Signal` or `Outcome`, the implementation of an Action on receipt of a `Signal`, and the state transitions applied by a `SignalSet` after processing of each of its `Signals`, are the responsibility of an individual HLS. The Activity service is part of the middleware that enables the HLS to operate in normal processing.

The same is true during recovery from any type of failure, be that application-related , hardware-related or network-related. Cooperation is required between the Activity service and the HLS and potentially between the HLS and the application driving it.

The HLS is responsible for deciding at which point, if any, during an Activity the participants need to be made persistent and notifies the `ActivityCoordinator`, through its `setPersistent` method, that the Activity has become recoverable. The HLS is responsible, following any failure between this point and the end of the Activity, for driving the `UserActivity recreate` method and then subsequently completing the Activity. The HLS is also responsible for persistently recording the essential state of its `Actions` and `SignalSets`. The Activity service is responsible for reconstructing the distributed `ActivityCoordinator` tree following recreation of a failed persistent Activity. The Activity service is not responsible for recovering any HLS state but rather calls the HLS `ServiceManager`, during recreation of an Activity, to obtain the recovered `SignalSets` and `Actions`.

4.5 Object Interactions

This section describes some typical Activity service object interaction sequence diagrams. This interactions are intended to be illustrative rather than prescriptive.

4.5.1 HLS initialization

An HLS `ServiceManager` is registered with a `UserActivity` object before that `UserActivity` object begins any Activities. This need happen only once, during HLS initialization.

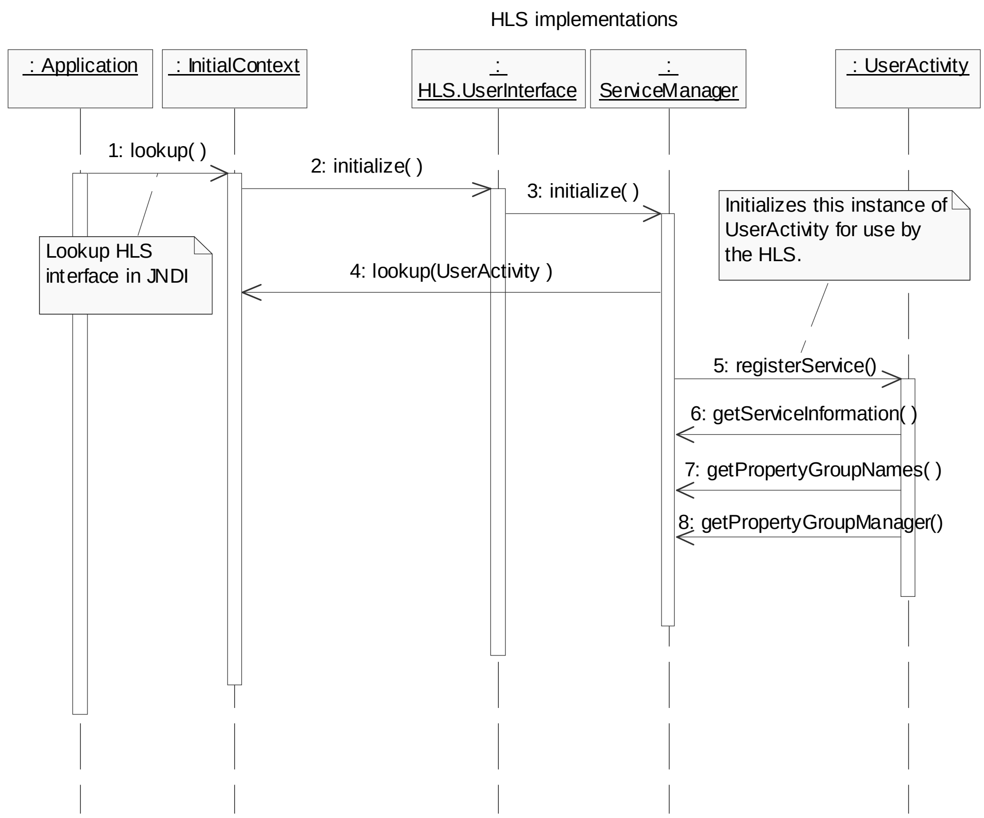

FIGURE 10 HLS initialization sequence diagram.

1. An application performs a JNDI lookup of the application interface provided by a HLS it uses.
2. An instance of the HLS user interface object bound in JNDI is created and initialized.
3. The HLS user interface object initialization obtains a reference to its `javax.activity.coordination.ServiceManager` interface.
4. The `ServiceManager` obtains the `UserActivity` reference from JNDI.
5. The `ServiceManager` initializes the `UserActivity` instance by registering itself via `registerService`.
6. The `UserActivity` instance completes its initialization by obtaining further information from the HLS `ServiceManager` - specifically the `ServiceInformation`...
7. ...list of `PropertyGroup` names used by the HLS...
8. ...and a `PropertyGroupManager` reference for each of these `PropertyGroups` from which instances of the `PropertyGroups` may be managed (created, related to parent and so on).

4.5.2 Begin an Activity

An applciation performs an operation that causes the HLS to begin an Activity.

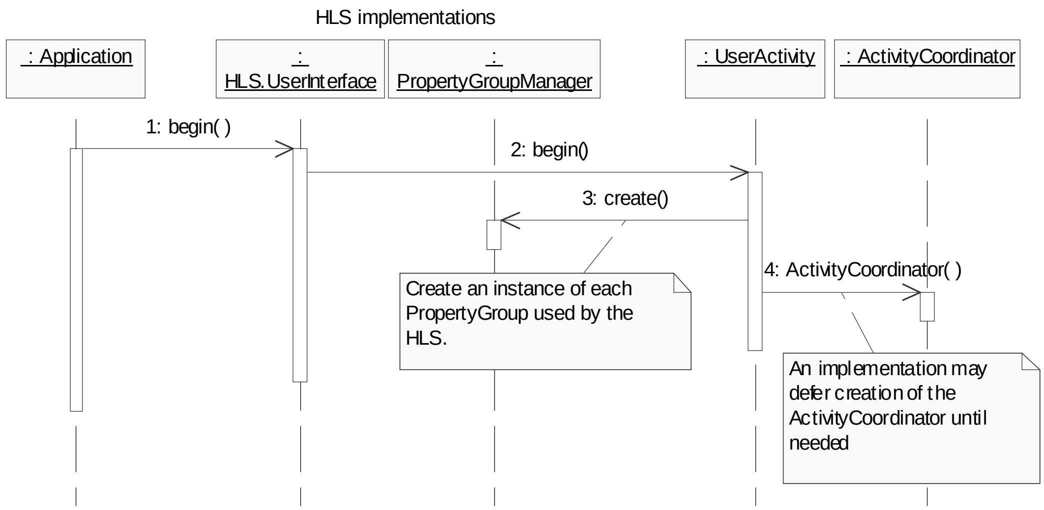

FIGURE 11 **Activity begin - sequence diagram**

1. An application starts a new HLS activity.
2. The HLS begins a new `UserActivity`.
3. The Activity service requests creates an instance of each `PropertyGroup`, used by the HLS, through its `PropertyGroupManager.`
4. An `ActivityCoordinator` instance is created for the new Activity. The `ActivityCoordinator` may not be needed until an Action is registered with the Activity, so this step may be deferred or eliminated in some Activities.

4.5.3 Add an Action

An application performs an operation that causes the HLS to register an Action with the Activity, with an interest in the specific SignalSet. Although this example doesn't show it, an Action may have have an interest in more than one SignalSet.

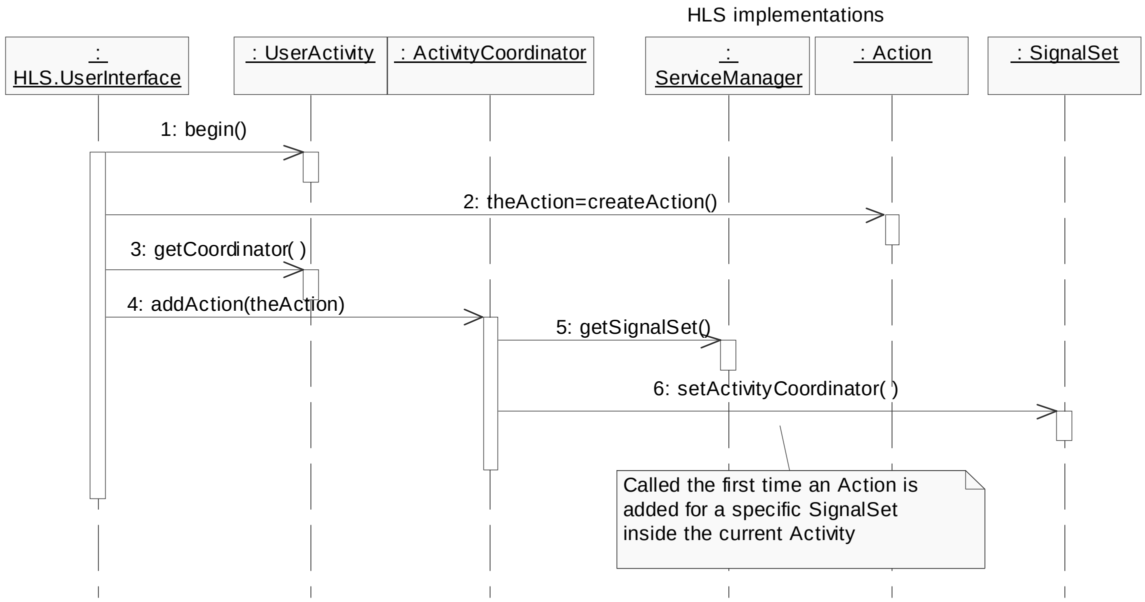

FIGURE 12 Add an Action - sequence diagram

1. The HLS begins an Activity.
2. The HLS creates an HLS Action using an mehanism specific to the HLS.
3. The HLS obtains the `ActivityCoordinator` from the `UserActivity` object.
4. The HLS registers the `Action` with the Activity service by passing it as a parameter on an `addAction` call, indicating which `SignalSet` the `Action` is interested in.
5. The `ActivityCoordinator` obtains a `SignalSet` instance from the `ServiceManager` if it isn't already using that `SignalSet` within the Activity.
6. The `ActivityCoordinator` passes a reference to itself to the `SignalSet` instance; this may be required by the `SignalSet` if it needs to make the Activity recoverable. At that time the `SignalSet` must call the `ActivityCoordinator setPersistent` method.

4.5.4 Complete and Activity

An application performs an operation that causes the HLS to complete the current Activity.

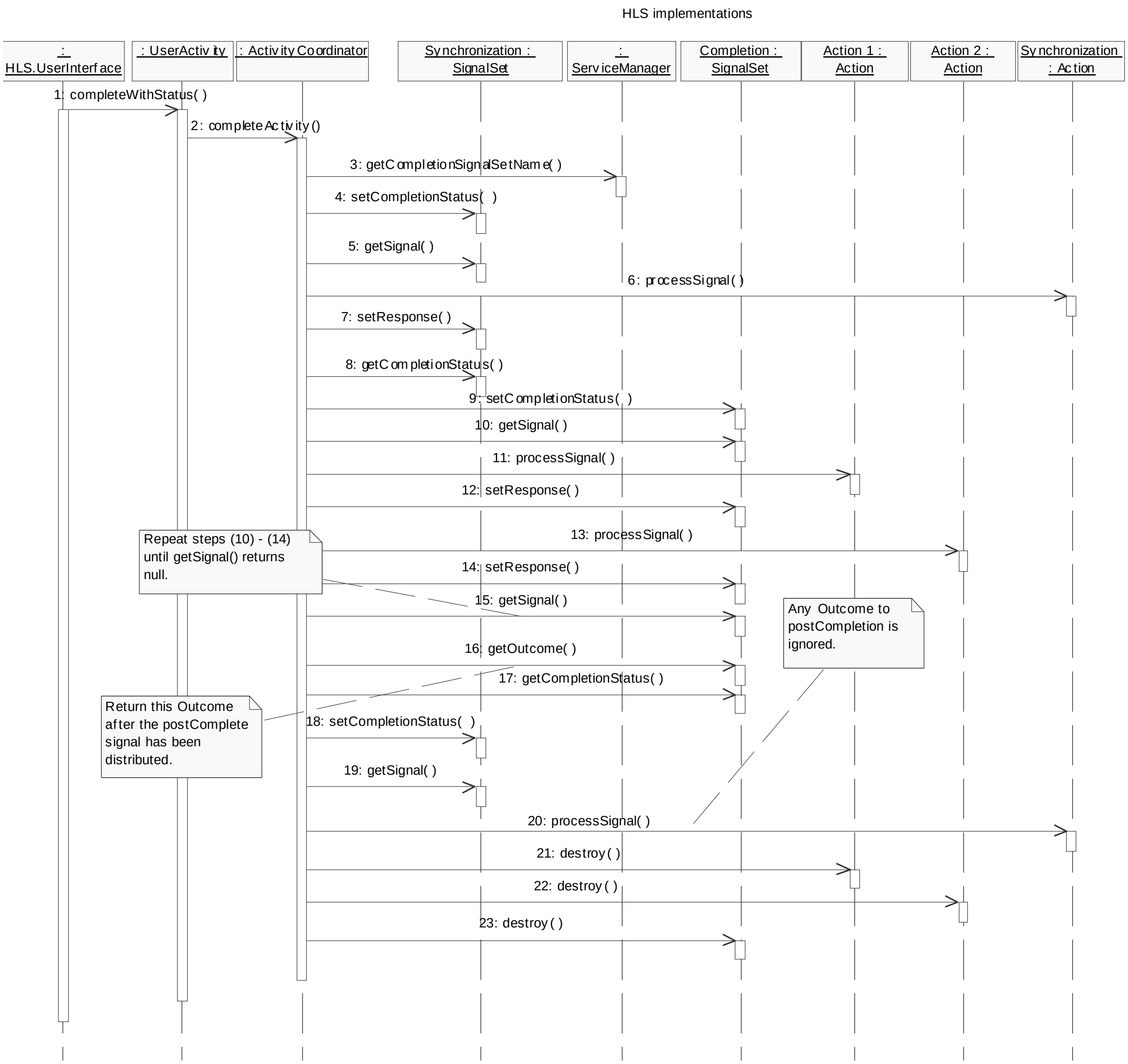

FIGURE 13 **Activity completion - sequence diagram**

1. An HLS peforms `completeWithStatus`, passing a `CompletionStatus`, for example `CompletionStatusSuccess`.
2. The `UserActivity` object instructs the `ActivityCoordinator` to complete the Activity. The specific `ActivityCoordinator` method to achieve this is internal to Activity service implementation - `completeActivity` is used in the figure for illustrative puposes.

3. If no specific completion `SignalSet` has been specified to the `ActivityCoordinator,` then the name of the default completion `SignalSet` can be obtained from the HLS `ServiceManager.` Alternatively, the completion `SignalSet` name could be obtained from the `ActivityCoordinator`. The `SignalSet` itself will already be in-use by the Activity if any `Actions` have been registered with an interest in it.
4. Processing of the predefined *Synchronization* `SignalSet` now begins; the Activity `CompletionStatus` is passed to the *Synchronization* `SignalSet`.
5. The first signal (`preCompletion`) is obtained from the *Synchronization* `SignalSet` if the `CompletionStatus` is `CompletionStatusSuccess`.
6. The `Signal` is sent to the highest priority `Action` that registered an interest in *Synchronization*, which returns an `Outcome` response. The Activity context is available on the thread during the processing of the signal.
7. This response is passed to the `SignalSet`; the `SignalSet` decides what to do next based on this response and returns a `CoordinationInformation` object. The `CoordinationInformation` object indicates whether the `preCompletion` signal should continue to be distributed to any remaining `Actions`.
8. Once `preCompletion` signaling is complete, the `ActivityCoordinator` obtains the updated `CompletionStatus` from the *Synchronization* `SignalSet`.
9. It sets this `CompletionStatus` into the completion `SignalSet`, to influence the completion `Signals` produced.
10. The first `Signal` is requested from the completion `SignalSet`.
11. The `ActivityCoordinator` sends this signal to the highest-priority `Action` interested in completion and obtains an `Outcome` from that `Action`. The Activity context is available on the thread during the processing of the completion signals.
12. The `ActivityCoordinator` passes this `Outcome` to the `SignalSet` which factors this `Outcome` into its state table and returns a `CoordinationInformation` object that indicates whether to continue sending the current `Signal` and whether to continue involving the current `Action`.
13. Assuming the `CoordinationInformation` object does not indicate that the current `Signal` should be abandoned, the `Signal` is sent to the next `Action`.
14. Again, the Action's `Outcome` is fed into the `SignalSet` and a `CoordinationInformation` object returned.
15. If the `CoordinationInformation` object indicates that the next `Signal` should be retrieved or if the previous `Signal` has been sent to all the interested `Actions`, then the `ActivityCoordinator` retrieves the next `Signal` from the `SignalSet`.
16. If the returned `Signal` reference is null, then the `SignalSet` has completed processing and the Activity service retrieves the final `Outcome` from the `SignalSet`. This `Outcome` will be returned on the `UserActivity` complete method that ultimately triggered the completion.
17. The `ActivityCoordinator` updates its view of the `CompletionStatus` from the `SignalSet`.
    (Not shown in the sequence diagram). Any `PropertyGroups` used by the HLS are called with `suspended` and then `completed`. The Activity context of the completing Activity is logically suspended prior to these calls on the `PropertyGroups`.
18. The final `CompletionStatus` is passed to the *Synchronization* `SignalSet`.
19. The `ActivityCoordinator` retrieves the `postCompletion` signal from the *Synchronization* `SignalSet`.

20. It sends this to all `Actions` registered with an interest in *Synchronization*. Any `Outcomes` from these `Actions` are ignored and cannot influence the `Outcome` of the Activity. The `postCompletion Signal` indicates that no further `Signal` will be sent to the `Action`, so it should destroy itself on completion of processing this `Signal`.
21. Actions that are not registered with the *Synchronization* `SignalSet` get explicitly told to `destroy` themselves at the end of the Activity. In this case, *Action 1* is destroyed.
22. *Action 2* is destroyed.
23. Finally, the completion `SignalSet` is told to `destroy` itself. After this, the `Outcome` produced in (16) is returned to the caller.

4.5.5 Broadcast Signals from a SignalSet

An application performs an operation that causes the HLS to broadcast a `Signal` to all interested `Actions` in the middle of an Activity. An HLS may support any number of `SignalSets` and may broadcast `Signals` from zero, one or more of those `SignalSets` during the course of the Activity.

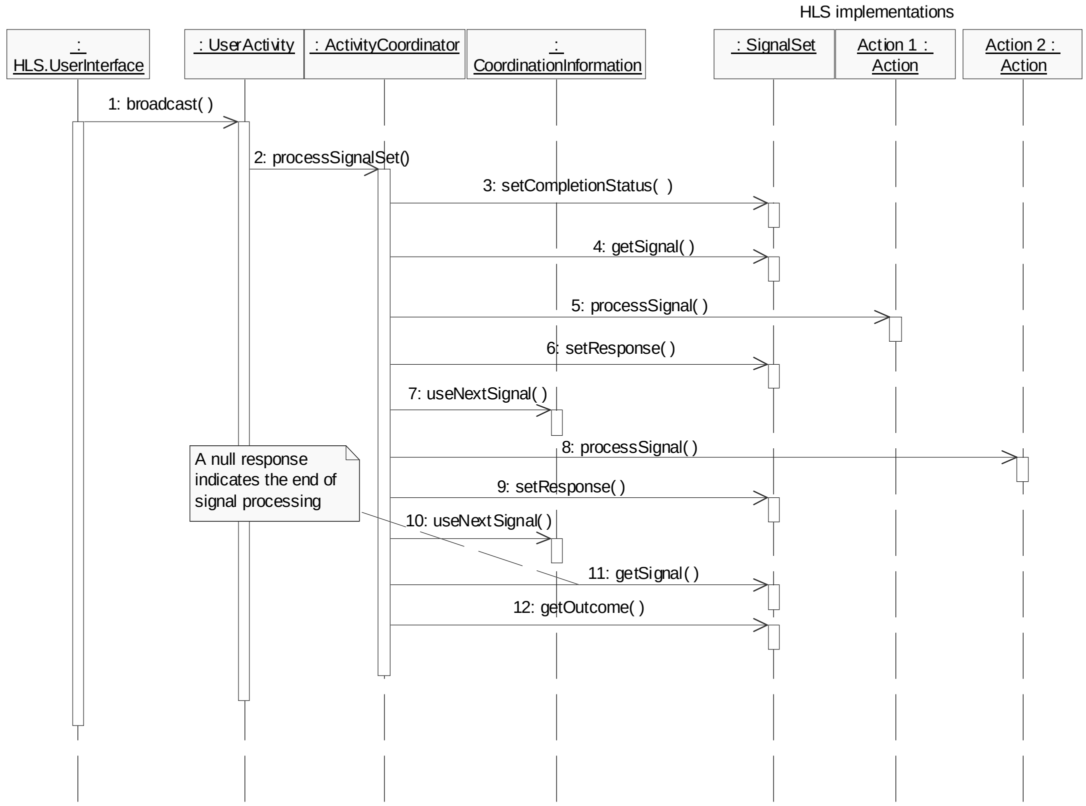

FIGURE 14 Broadcast - sequence diagram

1. An HLS wishes to broadcast `Signals` from a particular `SignalSet` to `Actions` with an interest in that `SignalSet` prior to completion, and does so by calling the `UserActivity` `broadcast` method.
2. The `UserActivity` object instructs the `ActivityCoordinator` to distribute the Signals of the specific `SignalSet`. The specific `ActivityCoordinator` method to achieve this is internal to Activity service implementation - `processSignalSet` is used in the figure for illustrative puposes.
3. The `ActivityCoordinator` indicates to the `SignalSet`, through the `setCompletionStatus` call, that the Activity is still active and is not completing.
4. The first `Signal` is requested from the `SignalSet`.
5. The `ActivityCoordinator` sends this signal to the highest-priority `Action` interested in the `SignalSet` and obtains an `Outcome` from that `Action`.
6. The `ActivityCoordinator` passes this `Outcome` to the `SignalSet` which factors this `Outcome` into its state table and returns a `CoordinationInformation` object
7. The `ActivityCoordinator` enquires of the `CoordinationInformation` object whether to continue sending the current `Signal` and whether to continue involving the current `Action`.

8. Assuming the `CoordinationInformation` object does not indicate that the current `Signal` should be abandoned, the `Signal` is sent to the next `Action`.
9. Again, the Action's `Outcome` is fed into the `SignalSet` and a `Coordination-Information` object returned.
10. The `ActivityCoordinator` enquires of the `CoordinationInformation` object whether to continue sending the current `Signal` and whether to continue involving the current `Action`.
11. If the `CoordinationInformation` object indicates that the next `Signal` should be retrieved or if the previous `Signal` has been sent to all the interested `Actions`, then the `ActivityCoordinator` retrieves the next `Signal` from the `SignalSet`.
12. If the returned `Signal` reference is null, then the `SignalSet` has completed processing and the Activity service retrieves the final `Outcome` from the `SignalSet`. This `Outcome` is returned on the `UserActivity broadcast` method.

4.5.6 Import an ActivityContext

The following sequence illustrates the processing of an inbound IIOP Activity service context. In the case of a received IIOP service context the service context filter is a Portable Interceptor, as defined by the CORBA specification[4]. An Activity service implemementation must provide a Portable Interceptor implementation in order to be able to process Activity-related work distributed over IIOP.

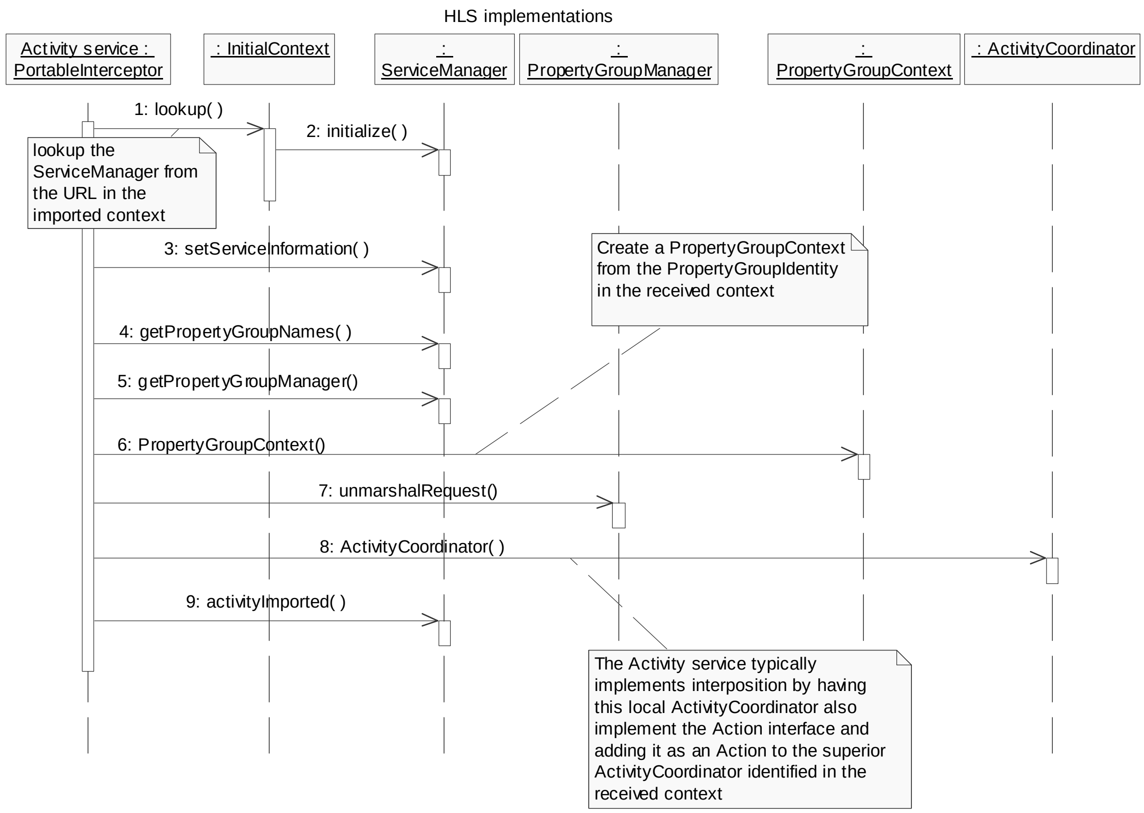

FIGURE 15 **Import an Activity service context - sequence diagram**

1. An inbound IIOP request is processed by an ORB and the registered Activity service portable interceptor's `receive_request` method is driven. If the request contains an Activity service context, the interceptor unmarshals it and examines the `activity_specific_data` of each `ActivityIdentity` to determine the lookup name of the `ServiceManager` for that `ActivityIdentity`. It performs a JNDI lookup of the `ServiceManager` name to obtain a `ServiceManager` object.
2. A `ServiceManager` instance is returned.
3. The `ServiceInformation` retrieved from the `activity_specific_data` is passed to the `ServiceManager`.
4. The interceptor retrieves the list of `PropertyGroup` names supported by the `ServiceManager`.
5. The interceptor requests an instance of a `PropertyGroupManager`, from the `ServiceManager`, for each type of `PropertyGroup` supported.
6. The interceptor creates a `PropertyGroupContext` object from each `PropertyGroupIdentity` structure contained within each `ActivityIdentity`.
7. The interceptor passes the `PropertyGroupContext` for each `PropertyGroup` to the appropriate `PropertyGroupManager` to unmarshal the `PropertyGroup` data.

8. The interceptor determines whether the received Activity context is already active within the receiving server and, if so, associates that context with the current thread. If it is not already active, the interceptor may create a new `ActivityCoordinator` and register it back as an `Action` with the superior (ie calling) node's `ActivityCoordinator` (ie it may interpose a local, subordinate ActivityCoordinator). As a standard performance optimization, the creation of an interposed `ActivityCoordinator` may be deferred until an `Action` is registered locally or an `ActivityContext` needs to be marshaled for an outbound request.
9. If a new context has been received from another domain then the `ServiceManager` is informed of this. This gives the HLS an *interception* point when a new context for the target `ServiceManager` is imported into the server.

4.5.7 Subordinate completion of an Activity

A subordinate `ActivityCoordinator` is registered as an `Action` with its superior `ActivityCoordinator`. The `Action` is registered with an interest in the pre-defined *Synchronization* `SignalSet` as well as any `SignalSets` that locally-registered `Actions` have an interest in.

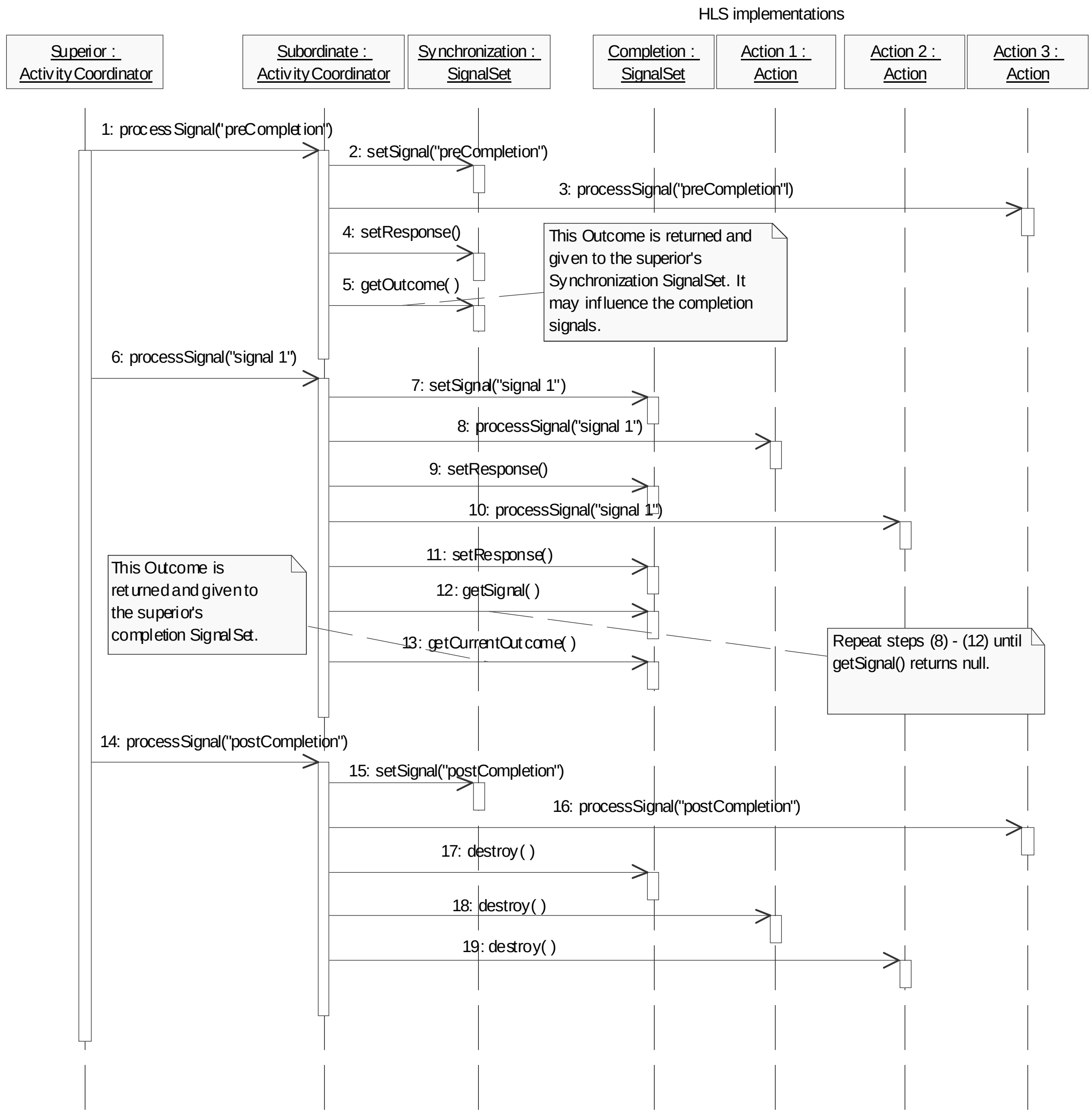

FIGURE 16 Subordinate completion - sequence diagram

1. The subordinate `ActivityCoordinator-Action` receives a `preCompletion` signal from its superior.
2. It calls `setSignal` on its local *Synchronization* `SignalSet` to indicate that a superior has produced this `Signal`.
3. The subordinate `ActivityCoordinator` then sends this `Signal` to the highest-priority `Action` that registered an interest in *Synchronization*, which returns an `Outcome` response.

4. This response is passed to the `SignalSet`; the `SignalSet` decides what to do next based on this response and returns a `CoordinationInformation` object. The `CoordinationInformation` object indicates whether the `preCompletion` signal should continue to be distributed to any remaining `Actions`.
5. Once `preCompletion` signaling is complete, the subordinate `ActivityCoordinator` obtains a correlated `Outcome` from the *Synchronization* `SignalSet` and returns it to its superior. This `Outcome` may affect the final `CompletionStatus` reached by the root *Synchronization* `SignalSet` and therefore the completion `Signals` produced by the root completion `SignalSet`.
6. The subordinate `ActivityCoordinator-Action` receives a completion `Signal` from its superior.
7. It calls `setSignal` on its local completion `SignalSet` to indicate that a superior has produced this `Signal`.
8. The subordinate `ActivityCoordinator` then sends this `Signal` to the highest-priority `Action` that registered an interest in the completion `SignalSet`, which returns an `Outcome` response.
9. This response is passed to the `SignalSet`; the `SignalSet` decides what to do next based on this response and returns a `CoordinationInformation` object. The `CoordinationInformation` object indicates whether the `Signal` should continue to be distributed to any remaining `Actions` and whether the called `Action` should receive any further `Signals`.
10. Assuming the `CoordinationInformation` object does not indicate that the current `Signal` should be abandoned, the `Signal` is sent to the next `Action`.
11. Again, the Action’s `Outcome` is fed into the `SignalSet` and a `CoordinationInformation` object returned.
12. If the `CoordinationInformation` object indicates that the next `Signal` should be retrieved or if the previous `Signal` has been sent to all the interested `Actions`, then the `ActivityCoordinator` retrieves the next `Signal` from the `SignalSet`.
13. If the returned `Signal` reference is null, then the `SignalSet` has completed processing of the received `Signal` and requires the next `Signal` to be produced by the superior `SignalSet`. The `ActivityCoordinator-Action` retrieves the current `Outcome` from the `SignalSet` and returns this to its superior, which processes the `Outcome` as it would an `Outcome` from any other `Action`.
14. After all the completion `Signals` have been produced, the root `ActivityCoordinator` drives the `postCompletion` `Signal`, which the subordinate `ActivityCoordinator-Action` has an interest in.
15. It calls `setSignal` on its local *Synchronization* `SignalSet` to indicate that a superior has produced this `Signal`.
16. The subordinate `ActivityCoordinator` then sends this to all `Actions` registered with an interest in *Synchronization*. Any `Outcomes` from these `Actions` is ignored and cannot influence the `Outcome` of the Activity. The `postCompletion` `Signal` indicates that no further `Signal` will be sent to the `Action`, so it should `destroy` itself on completion of processing this `Signal`.
17. The local completion `SignalSet` is told to `destroy` itself.
18. `Actions` that are not registered with the *Synchronization* `SignalSet` get explicitly told to `destroy` themselves at the end of the Activity.
19. `Action` 2 is destroyed.

4.5.8 *Beginning a child Activity*

An application performs an operation that causes the HLS to begin an Activity as a child of an existing Activity. Actions registered with the parent Activity's *ChildLifetime* `SignalSet` are informed of this event.

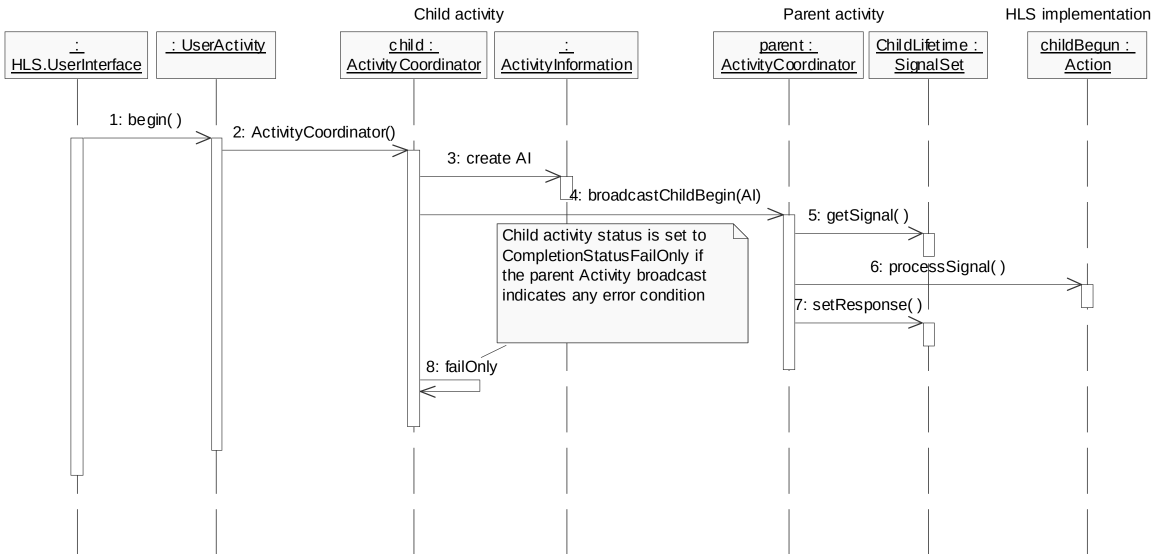

FIGURE 17 Beginning a child Activity - sequence diagram

1. The HLS begins an Activity, while an Activity is already active on the thread, by calling `UserActivity` in the usual way. This Activity is the child of the existing Activity.
2. The Activity service creates its internal objects, including the `ActivityCoordinator`, as described in "Begin an Activity" on page 30.
3. The Activity service creates an `ActivityInformation` object describing the child Activity.
4. The `UserActivity` object notifies the parent's `ActivityCoordinator` that the child Activity has begun, passing the child Activity's `ActivityInformation`. The specific `ActivityCoordinator` method to achieve this is internal to the Activity service implementation - broadcastChildBegun is used for illustrative purposes.
5. In the parent Activity, processing of the predefined *ChildLifetime* `SignalSet` begins if any Actions have been registered with the parent `ActivityCoordinator` with an interest in *ChildLifetime* signals. The `ActivityCoordinator` retrieves the first (and only) signal (*childBegin*) from the *ChildLifetime* `SignalSet`.
6. The `Signal` is distributed to each registered `Action`.
7. There are no predefined `Outcomes` for the *ChildLifetime* `SignalSet` so the only response that would be set would be following an exception during signal processing.
8. Any failure in the parent processing the *ChildLifetime* `SignalSet` results in the child Activity `CompletionStatus` being set to `CompletionStatusFailOnly`.

4.5.9 *Recovering after failure*

An HLS recreates an Activity that was previously made persistent. Any recovery of the HLS itself is outside the scope of the Activity service specification. The HLS is responsible for initiating the recreation of the persistent Activity following a failure. The HLS must also complete the Activity at some stage after recreating it.

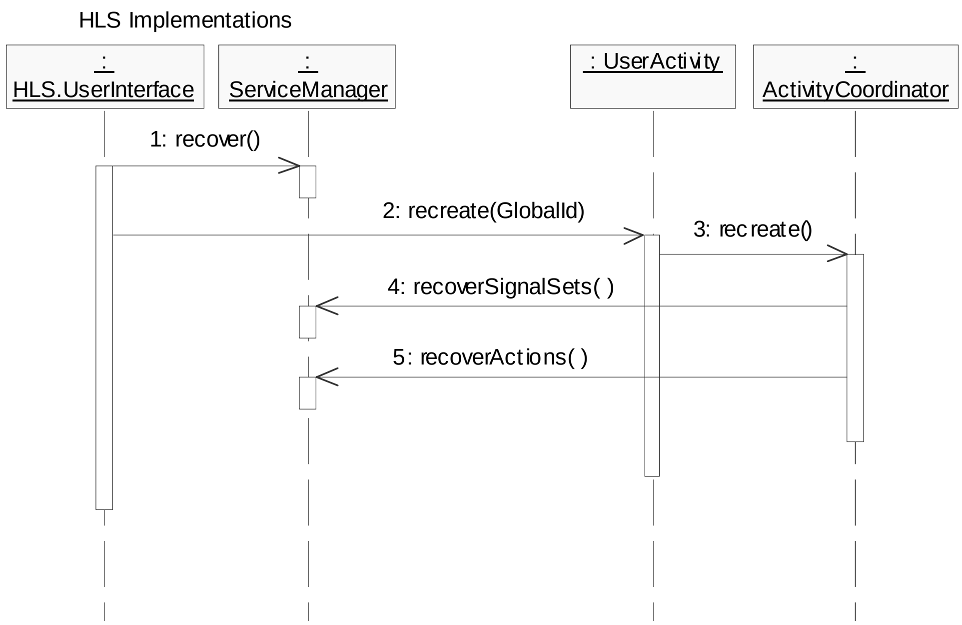

FIGURE 18 Recovery after failure - sequence diagram

1. After performing any required initialization (for example, as described in “HLS initialization” on page 28) the HLS performs a service-specific recovery of its own data, including all incomplete and persistent `GlobalIds`.
2. For each recovered `GlobalId`, the HLS calls `recreate` on the `UserActivity` object.
3. The Activity service creates an `ActivityCoordinator` object for the Activity being recreated. The specific mechanism for doing this is internal to the Activity service implementation - the recreate call on the `ActivityCoordinator` is used here for illustrative purposes.
4. The Activity service retrieves the list of HLS `SignalSets` used in the Activity from the HLS `ServiceManager`.
5. For each recovered `SignalSet` the Activity service retrieves the list of `Actions` registered for that `SignalSet` and re-registers them with the recovered `SignalSet`.

5.0 Interoperability

5.1 Requirements on an Activity service implementation

A J2EE Activity service implementation is required to be interoperable across different vendors' ORB boundaries. The format of the interoperable service context is defined by the `org.omg.CosActivity.ActivityContext` structure in the OMG Activity service specification. The IOR for any object that supports the receipt of Activity service context must have an `org.omg.CosActivity.ActivityPolicy` value of REQUIRES or ADAPTS encoded in the TAG_ACTIVITY_POLICY of the IOR.

A J2EE Activity service implementation is required to be interoperable with a CORBA Activity service implementation so long as the latter implements interposition; that is, the CORBA Activity service must create a local `org.omg.CosActivity.ActivityCoordinator` when inbound Activity context is received from an upstream (superior) node and register an `org.omg.CosActivity.Action` back to the superior's ActivityCoordinator (whose reference is passed in the `ActivityContext` service context).

A J2EE Activity service must implement interposition by creating a local `javax.activity ActivityCoordinator` when inbound Activity context is received from an upstream (superior) node and registering an `org.omg.CosActivity.Action` back to the superior's `org.omg.CosActivity.ActivityCoordinator`.

A J2EE Activity needs to propagate information, in the Activity service context, to identify the type of HLS that created the Activity in order that the appropriate `ServiceManager` be located in the target system. The `org.omg.CosActivity.ActivityIdentity` structure in the Activity service context contains an unsigned long `type` field, which for J2EE Activities must be set to `0x4A324545` by the domain that creates the service context. For such Activities, the `activity_specific_data` field of the `ActivityIdentity` is architected to encode information specific to J2EE HLS's. The `activity_specific_data` field is of type `org.omg.CORBA.Any` and, for a J2EE Activity type, contains a `TypeCode` with a `TCKind` of `_tk_struct` and a value of a `j2ee_activity_specific_data` structure which is defined in IDL as follows:

```
module javax
  {
    module activity
    {
      struct j2ee_activity_specific_data
      {
        string service_name;
        string context_group;
        any service_specific_data;
        any extended_data;
      };
```

```
    };
};
```

The `j2ee_activity_specific_data` structure is referenced, within the J2EE domain, through a `javax.activity.ServiceInformation` object. `ServiceInformation` data for an Activity is populated by the HLS `ServiceManager`. The data is consumed, in the target domain of a remote request, in part by the Activity service (to determine the `ServiceManager` to use) and in part by the `ServiceManager` implementation in that domain.

5.2 CORBA interfaces

A J2EE Activity service provider must provide implementations of the `org.omg.CosActivity.ActivityCoordinator` and `org.omg.CosActivity.Action` interfaces that satisfies the requirements for interoperability stated in 5.1 “Requirements on an Activity service implementation”, on page 43. These are internal to the implementation of the Activity service and need not be exposed to any J2EE Activity service HLS.

A J2EE Activity service provider may optionally support a configuration in which HLS-provided SignalSets are remote from the ActivityCoordinator. This may be desirable for some J2EE platforms in which *application* code (and an HLS-provider can be considered to be *application* code, although some HLS’s may become part of the *application server/container*) is deployed in a separate JVM from the *system* code (*e.g.* the Activity service implementation itself, which is part of the *application server/container*). Such a configuration would require the `ActivityCoordinator` implementation to be able to make calls to a (remote) `org.omg.CosActivity.SignalSet` and would require an implementation of an `org.omg.CosActivity.SignalSet` in the remote domain that passed these requests onto the local `javax.activity.coordination.SignalSet`.

5.3 CORBA Exceptions

The following CORBA System Exceptions have been added to CORBA 3.0 for use by the CORBA Activity service. Each of these requires an equivalent J2EE Activity service `java.rmi.RemoteException`:

INVALID_ACTIVITY -- maps to `javax.activity.InvalidActivityException`. This system exception may be thrown on any method for which Activity context is accessed and indicates that the attempted invocation or the Activity context associated with the attempted invocation is incompatible with the Activity's current state. It may also be thrown by a container if Activity context is received on a method for which Activity context is forbidden. This exception will be propagated across

ORB boundaries via an `org.omg.CORBA.INVALID_ACTIVITY` system exception. An application should handle this error by attempting to complete the Activity.

ACTIVITY_COMPLETED -- maps to `javax.activity.ActivityCompletedException`. This system exception may be thrown on any method for which Activity context is accessed and indicates that ongoing work within the Activity is not possible. This may be because the Activity has been instructed to complete with `CompletionStatusFailOnly` or has ended as a result of a timeout. This exception will be propagated across ORB boundaries via an `org.omg.CORBA.ACTIVITY_COMPLETED` system exception. An application should handle this error by attempting to complete the Activity.

ACTIVITY_REQUIRED -- maps to `javax.activity.ActivityRequiredException`. This system exception is thrown by a container if Activity context is not received on a method for which Activity context is mandatory. This exception indicates a deployment or application configuration error. This exception will be propagated across ORB boundaries via an `org.omg.CORBA.ACTIVITY_REQUIRED` system exception.

5.4 Behaviour in the case of unknown Activity types, ServiceNames or PropertyGroups

When an `ActivityContext` is received by a domain on which no Activity service is configured, the `ActivityContext` is ignored.

When an `ActivityContext` is received by a domain on which the Activity service is configured, the `ActivityContext` is processed according to the following rules:

- If the `org.omg.CosActivity.ActivityIdentity.type` or `activity_specific_data` are not recognized, an `InvalidActivityException` is thrown.
- If the `service_name` in the `activity_specific_data` is not recognized, then Activity context is resumed into the `context_group` defined within the `activity_specific_data` in order that the context nesting hierarchy is preserved on flows to downstream domains. The Activity context is otherwise not available to an HLS in the importing domain.
- If a `PropertyGroupIdentity` structure is received for which no local `PropertyGroupManager` is available, the `PropertyGroupIdentity` data is cached with the Activity in its marshalled form and will be propagated on flows to downstream domains. The `PropertyGroupIdentity` data is otherwise not available to an HLS in the importing domain.

6.0 Impact on other specifications

The following specifications will need to be modified to accommodate the J2EE Activity service.

EJB specification -- Under “Support for Distribution and Interoperability”, the table of mapped System Exceptions needs to be extended to include the following CORBA standard exceptions, introduced by the OMG Activity service specification.

J2EE exception	Mapped CORBA exception
javax.activity.InvalidActivityException	INVALID_ACTIVITY
javax.activity.ActivityCompletedException	ACTIVITY_COMPLETED
javax.activity.ActivityRequiredException	ACTIVITY_REQUIRED

TABLE 1 **New standard exception mappings**

--

Under “Exception handling”, 3 new EJBExceptions need to be defined with equivalent meanings to the 3 javax.activity RemoteExceptions. These are:

- javax.ejb.InvalidActivityLocalException
- javax.ejb.ActivityCompletedLocalException
- javax.ejb.ActivityRequiredLocalException

An EJB container may raise these exceptions in the event of a method dispatch to a EJB local interface failing for reasons of improper Activity context.

Java-to-IDL specification -- support the new exception mappings.

J2SE -- CORBA Activity service system exceptions (which are defined in CORBA 3.0) need to be supported by J2SE. These are part of a larger list of new CORBA 2.6 and CORBA 3.0 system exceptions that J2SE 1.4 does not currently know about. The inclusion of the new CORBA exceptions in J2SE will be persued through JSR 176 (J2SE 1.5)

OMG standard tags -- An `org.omg.CosActivity.ActivityIdentity.type` needs to be allocated for J2EE Activities. A value of `0x4A324545` has been requested. The format of the `ActivityIdentity.type` is described in 5.0 “Interoperability”, on page 43.

Appendix A Specific HLS examples

This section contains examples of high-level services that use the Activity service. Other examples may also be found in the OMG Activity service specification[2]. None of these examples are intended to be prescriptive - they merely show ways in which a HLS may be employed to exploit the facilities of the Activity service.

A.1 Long-running Unit of Work (LRUOW)

A.1.1 The Problem

As business processes execute concurrently over extended periods, it is increasingly likely that these processes will attempt to access the same data. To ensure data integrity and consistency, a concurrency control mechanism is needed, and transactions are often employed for this purpose.

A.1.2 A simple example

Figure 19 illustrates a simple business process.

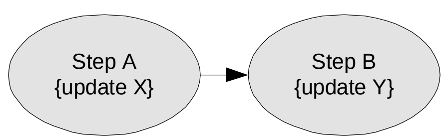

FIGURE 19 Simple business process

This process has two steps. *Step A* might involve, for example, updating some object *X*, and *step B* might be involve updating some object *Y*. In this case, we would like to execute the entire process as a single atomic action: either both X and Y are updated by the process, or neither.

We could execute both steps within a single global transaction, as illustrated in below in Figure 20.

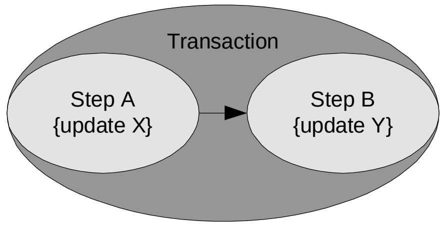

FIGURE 20 Business process in global transaction

For long-lived processes, however, there are some problems with this approach. First, if the process is long-lived, it introduces a temporal dependency between the two steps. In our example, it requires that object X and object Y be available at the same time. Such a dependency might not be desirable, or even possible, depending on the system. Furthermore, such temporal dependencies can potentially reduce concurrency; for example, locks acquired in step A must be held until after step B completes. A conseqeunce of considering this process as a global transaction is that the entire process is a unit of failure: that is, if one step fails, all steps fail. If step A is costly, we might like to avoid re-doing it once it executes successfully.

Consider splitting each step into a separate transaction as illustrated below in Figure 21.

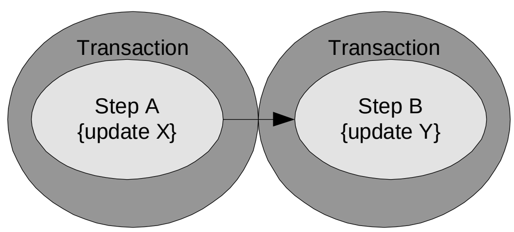

FIGURE 21 Transactional steps

If, after executing step A, step B cannot be executed straight away (because object Y is not available, for example), the step can be rescheduled for a later time. In doing so, the process becomes forward recoverable: if step B fails, we need not redo step A.

The problem with this approach is that the process is no longer atomic. For example, what if we update object X (in step A) but cannot reach object Y (in step B)? We can no longer simply abort a global transaction to reverse the effect of step A. Another problem is that the process no longer executes in isolation; that is, partial effects are visible before the entire process completes (e.g., there is a point where the update to object X is visible but not the update to object Y.).

To ensure process atomicity, we could use *compensation steps*, as proposed in the Sagas, transactional workflows and Open-nested Transaction models. A compensation step is used to reverse the effect of some previously executed step (or steps) within a partially completed process. In Figure 22, for example, step B cannot be executed, and the overall process must be aborted. Having successfully executed step A, compensation step *A'* is used to reverse its effect, returning the system to semantically equivalent initial state.

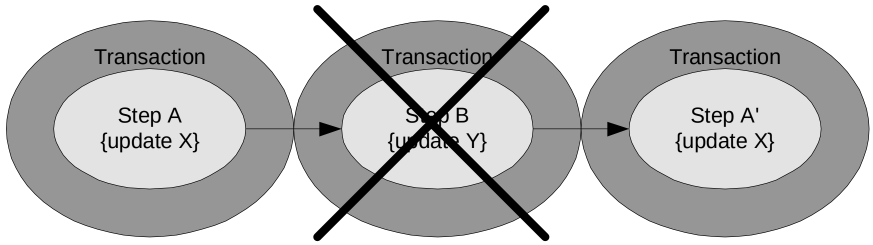

FIGURE 22 Compensation steps

One consideration with this approach is that compensation logic can become complex, and difficult to get right. Further, there is limited support for compensation in existing commercial systems.

A.1.3 The LRUOW Approach

The Long-running Unit of Work (LRUOW) approach[5] addresses these problems by allowing a long-running business process to execute as multiple, transactional steps, while providing isolation and atomicity for the process as a whole.

Figure 23 shows business process executing within the context of an LRUOW. Process steps execute as transactions within the LRUOW context. Prior to completion of the process, the effects of its steps are visible only to other steps executing within that same process (i.e., the same LRUOW context). Steps executing concurrently within the same process execute as concurrent transactions, and interact according to the semantics of the underlying transaction model.

5. B. Bennett, B. Hahm, A. Leff, T. Mikalsen, K. Rasmus, J. Rayfield, and I. Rouvellou. "A Distributed Object Oriented Framework to Offer Transactional Support for Long Running Business Processes" in J. Sventek and G. Coulson, editors, *Proceedings IFIP/ACM International Conference on Distributed Systems Platforms*, New York , NY, USA April 2000 (Middleware 2000); Lecture Notes in Computer Science 1795, Springer-Verlag, Berlin, pp. 331-348, April 2000.

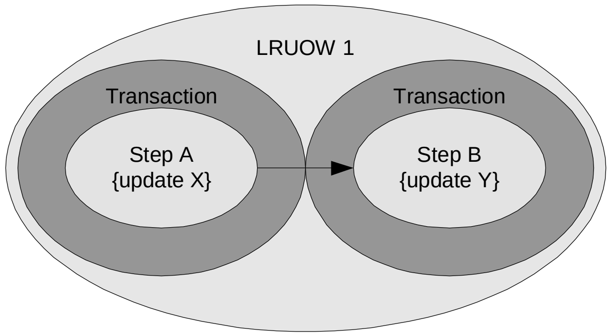

FIGURE 23 **LRUOW business process**

A.1.3.1 Version spaces

Each LRUOW context creates a *version space*, which provides isolation for the LRUOW: the effects of steps executing in one LRUOW are independent of a steps executing in other LRUOWs.

Steps executing within a LRUOW context represent transactions that transform a version space between consistent states. The initial state of a version space is the state of the *global version space* at the time that the LRUOW context is created. When a LRUOW completes successfully, its version space must be *reconciled* with the current state of the global version space.

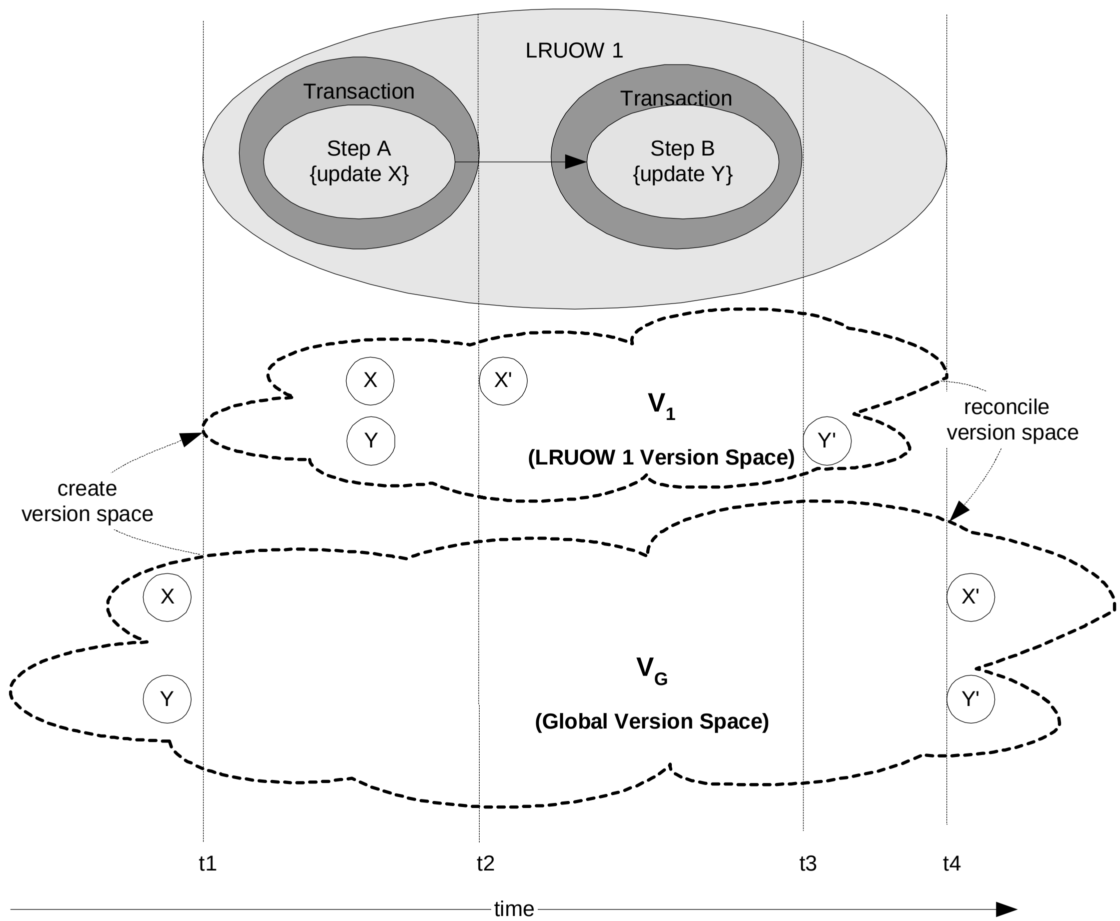

FIGURE 24 **Version spaces**

Figure 24 shows an LRUOW, its associated version space, and the global version space:

- At time *t1*, context LRUOW 1 and associated version space V_1 are created. The initial state of V_1 is the state of the global version space, V_G, at time *t1*: $V_1 = V_G = \{ X , Y \}$.
- At time *t2*, step A's update to object X is committed: $V_1 = \{ X', Y \}$.
- At time *t3*, step B's update to object Y is committed: $V_1 = \{ X' ,Y' \}$.
- At time *t4*, the LRUOW completes, and changes to version space V_1 are reconciled with the global version space: $V_G = \{ X' , Y' \}$.

In this example, the global version space did not change during LRUOW 1's execution, so the reconciled state of V_G is simply V_1's final state. If, however, another LRUOW had completed before LRUOW 1 (but after LRUOW 1 begun), the reconciliation process would not have been so simple. We will return to this subject later in Section A.1.3.3, "Version Space Reconciliation".

A.1.3.2 Nesting

LRUOW contexts, and associated version spaces, can be nested, as is illustrated in Figure 25.

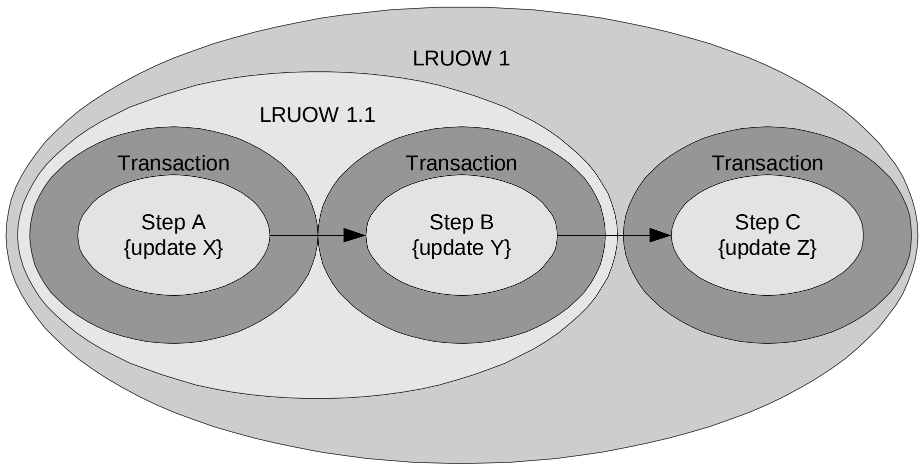

FIGURE 25 Nested LRUOWs

A.1.3.3 Version Space Reconciliation

The technique used to reconcile a version space (or subset of a version space) is an implementation detail of the *versioned resource managers* implementing the version space, as described later. As examples of existing techniques, two general methods have been proposed [5]: *Predicate Transform* (*P/T*) and *Conflict Detection and Resolution* (*CDR*).

Predicate/Transform (P/T):

This method associates a predicate with each transform to be applied to a version space. Before applying the transform, the associated predicate is evaluated within the context of the version space. If the predicate holds, the transform is applied to the version space, and the predicate/transform pair is logged. To reconcile this version space with the global version space, each logged predicate/transform pair is examined, in order. If the predicate still holds when evaluated within the context of the global version space (or parent version space, if nested), the transform is applied to the global version space. Otherwise, if the predicate no longer holds, version reconciliation fails, and the associated LRUOW can attempt to fix the problem.

Conflict detection / Resolution (CD/R)

This method proposes a structured mechanism for detecting conflicts between version space objects and applying algorithms that resolve such conflicts when they occur.

A.1.3.4 Transactional reconcile step

In some cases, version space reconciliation can fail. When this occurs, the business process can detect the failure and initiate additional steps and attempt to correct the problem. To avoid version space inconsistencies that could arise as part of such a failure, reconciliation is itself executed transactionally.

As shown in Figure 26, an implicit *reconcile step* is used to transactionally transform the global version space between consistent states.

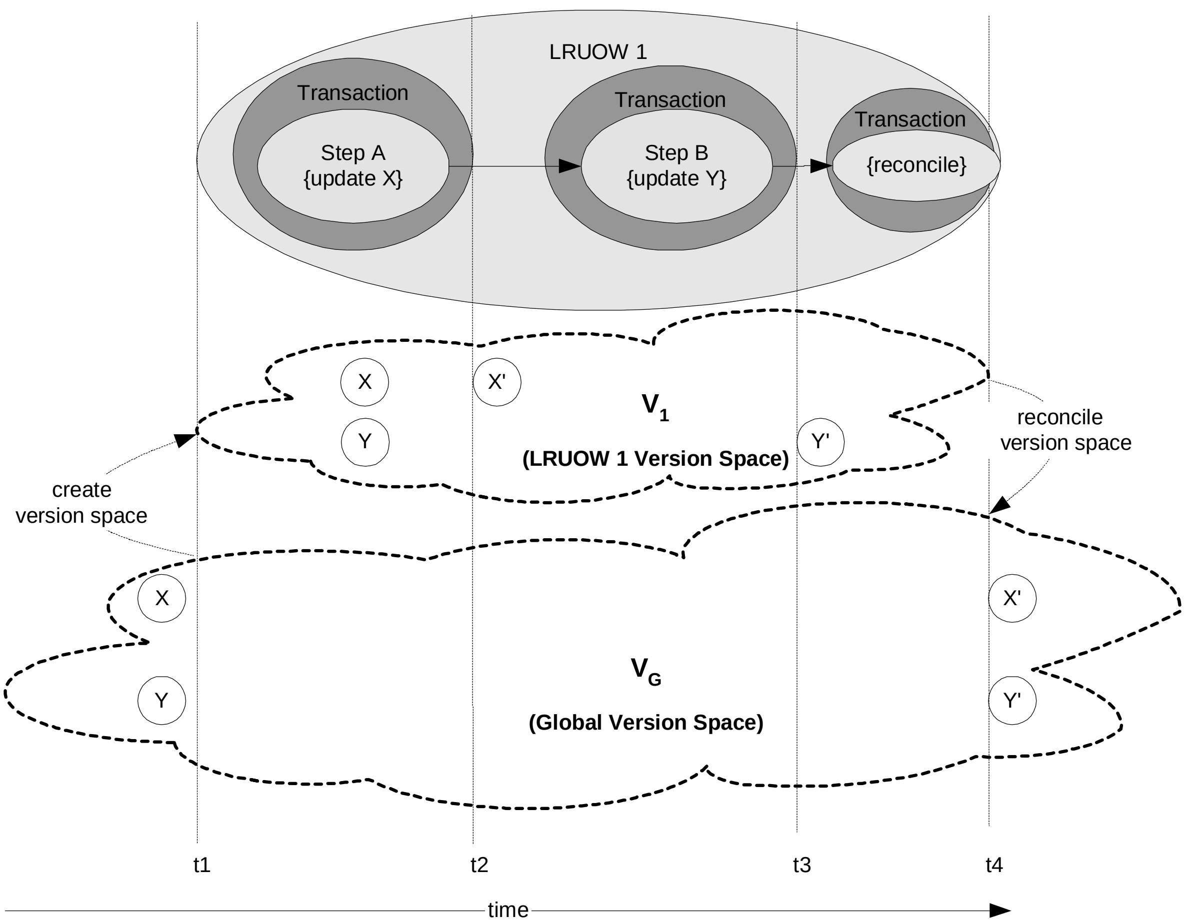

FIGURE 26 Transactional reconcile step

At time *t4*, the effects of the reconcile step are committed. Though the reconcile step updates the global version space (or parent version space, if nested), it is still considered to be part of the child LRUOW context.

A.1.4 LRUOW as a High-level Service

LRUOW contexts are modeled as *activities*, and each business process step executes as a *JTA transaction* subordinate to such an activity. This is illustrated in Figure 27.

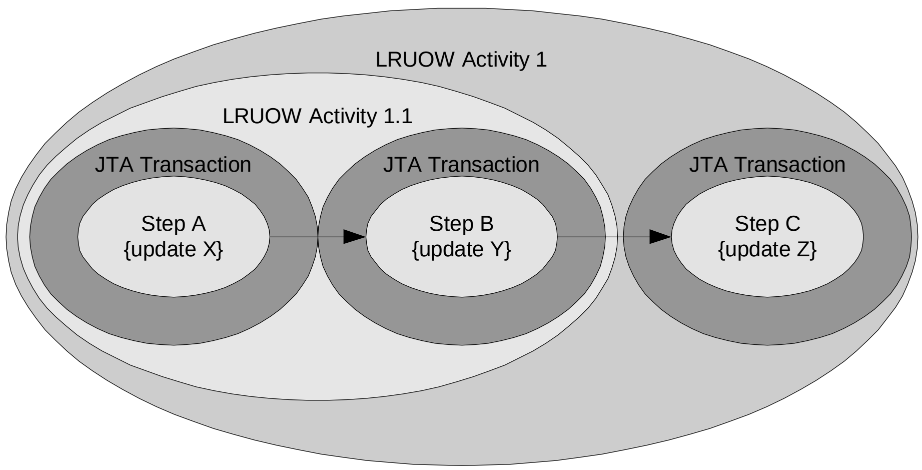

FIGURE 27 **LRUOW activities and transactions**

The completion processing of a LRUOW activity involves an implicit *reconcile step* which is used to reconcile changes in the LRUOW's version space with any changes in the global (or parent) version space. This is embodied in the LRUOW HLS by a `ReconcileSignalSet` producing signals for registered `ReconcileActions`. Like business process steps, the reconcile step executes as a JTA transaction subordinate to the LRUOW activity.

A.1.4.5 Versioned resource managers

Version spaces are modeled using *version resource managers* (VRMs). A VRM supports three interfaces: the *version space interface* (VSI), the *JTA XAResource interface* [JTA spec], and an application programming interface (API) (such as SQL).

Version space interface (VSI)

The LRUOW HLS service manager uses this interface to create and reconcile version spaces, and to associate application threads with version spaces. Below, we briefly present this interface:

vsi_start(vsid) - Start (or resume) work within a version space, given by *vsid*, associating the calling thread with the version space. Subsequent invocations (by the same thread) on the VRMs API execute within the context of the given version space.

vsi_end(vsid) - End (or suspend) work within a version space, given by vsid, removing the calling thread's association with the version space.

vsi_reconcile(vsid) - Rconcile work performed within a version space.

vsi_abandon(vsid) - Abandon work performed within a version space.

JTA XAResource interface

The LRUOW HLS service manager and transaction manager use this interface to associate application threads with transaction contexts, and to coordinate distributed transactions.

Application Program Interface (API)

Application steps use this interface to manipulate objects within version spaces.

When multiple VRMs are accessed as part of an LRUOW, each VRM represents a subset of the LRUOW's associated version space. Executing the reconcile step as a transaction ensures that the version space is reconciled as an atomic action. This is illustrated below in Figure 28.

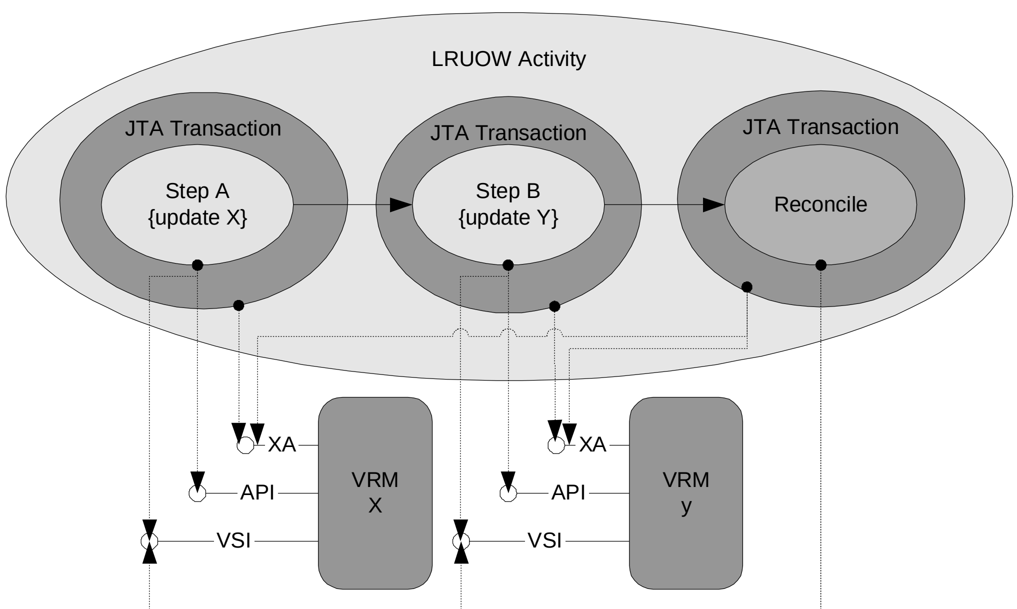

FIGURE 28 Atomic version space reconciliation

A.1.4.6 Reconcile SignalSet

The LRUOW HLS implements the `javax.action.coordination` package's `SignalSet` and `Action` interfaces through `ReconcileSignalSet` and `ReconcileAction` objects.

When first accessed, a VRM becomes a participant in the LRUOW activity by adding a `ReconcileAction` to the `ReconcileSignalSet` associated with the LRUOW's activity. To successfully complete the LRUOW, the ReconcileSignalSet produces `ReconcileSignals` for distribution to all registered `ReconcileActions` during completion processing.

A `ReconcileAction` is simply a proxy to an associated VRM. When a ReconcileAction receives the `ReconcileSignal`, it invokes the vsi_reconcile() method on its associated VRM. If successful (i.e., the version space was reconciled), the `ReconcileAction` returns an outcome of `ReconciledOutcome`; otherwise, an outcome of `ReconcileFailedOutcome` is returned.

The reconcile step is triggered by a request, on the part of the application, to complete an LRUOW. As mentioned above, the reconcile step executes transactionally; that is, the broadcast of `ReconcileSignals` to `ReconcileActions` occurs within a transaction. If any `ReconcileAction` fails (i.e., produces an outcome of `ReconciledFailedOutcome`) this transaction is rolled back, reverting the version space to its previously committed state. Otherwise, if all ReconcileActions succeed (i.e., produce an outcome of `ReconciledOutcome`), the transaction is committed.

A.1.4.7 VRM adapters

In a J2EE application server environment, application steps access VRMs indirectly through a *VRM adapters*. A VRM adapter is analogous to a JCA Resource Adapter [6] (and can be implemented as such).

When the application requests a connection, the VRM adapter first enlists the VRM in the appropriate version space and transaction (using the VSI and XAResource interfaces described above); this may involve adding `ReconcileAction's` as described above. Then, subsequent application requests through the connection execute within the correct context. When the application closes the connection, the VRM adapter delists the VRM from the LRUOW and transaction.

6. J2EE Connector Architecture, V1.0, *Sun Microsystems Inc*

A.2 Open Nested Transactions

This section describes the Open Nested Transaction (ONT) model, which is an example of an extended transaction model that is implemented as a High Level Service on top of the J2EE Activity service. In this model an Activity may contain any number of nested activities, which may recursively contain other nested activities organized into a hierarchical tree of nested activities or an *Activity family.*

Each activity or nested activity represents an atomic unit of work to be done; that is a JTA transaction. The creation of an activity or nested activity implies the creation of an associated top-level, or flat transaction, which may possibly contain nested transactions, if the provided Transaction Service supports nested transactions. From the application point of view, an activity is implicitly transactional, and this combined activity and transaction is referred to as an ONT Activity.

The ONT model respects the following rules:

- ONT Activities are strictly nested. An ONT Activity cannot complete with Success unless all of its children have completed. Since an ONT Activity is implicitly transactional, completing with Success means that the associated transaction is committed.
- When an ONT Activity completes with Failure, all of its children in an active state are completed with Failure. Completing with Failure means that the associated transaction is rolled back.
- When an ONT Activity completes with Failure (rolls back), all of its children which have completed with Success (committed) shall be compensated if compensating actions have been defined. The behavior of the compensation action is defined by the application since it is only the application that possesses sufficient information to do compensation.

A.2.5 The end-user programming interface

A new interface, *UserOpenNested*, is provided to allow end-users to control ONT Activities. This interface invokes the appropriate interfaces provided by the underlying JTA and Activity services, respectively, to manage transactions and to manage activities. The service that provides the UserOpenNested interface and that manages the underlying Activities and transactions is referred to as the *OpenNested Service*.

Propagation of an ONT Activity to a diferent execution environment is achieved through an Activity Context, to represent the Activity, and an OTS transaction context to represent the top-level transaction.

The OpenNested Service classes, interfaces and exceptions are described fully in the javadoc packages that accompany this specification.

A.2.5.1 ActivityStatus

The diagram below indicates the transitions an ONT Activity can undergo. The Open Nested Service defines Status values that are specific to this model, conveying more useful meaning that the more general Activity service Status values.

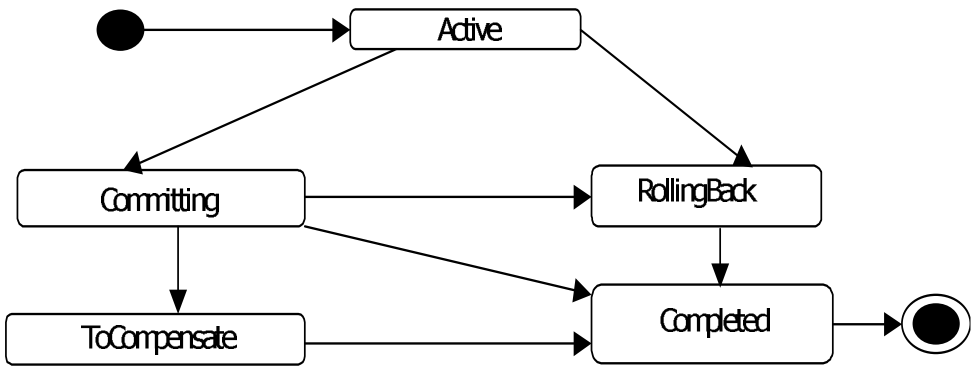

FIGURE 29. Transactional Activity and UML state diagram.

A.2.5.2 UserOpenNested interface

The **`UserOpenNested`** interface defines operations that allow the client to begin and end ONT Activities and to obtain information about the current Activity and associated transaction. The methods on this interface will generally make calls to the underlying JTA and Activity service interfaces.

Since this **`UserOpenNested`** interface aims to be layered on both JTA and the Activity Service, exceptions raised by those services are caught by this interface and re-raised to the end-user application.

How the UserOpenNested implementation is obtained is not mandated by this specification, but it could for example be obtained by performing a JNDI lookup of “java:comp/UserOpenNested” on the InitialContext.

A.2.5.3 Compensator interface

The **`Compensator`** interface is provided to define a generic mechanism to manage the compensating action of a committed ONT Activity if one of its ancestors has rolled back. It should be implemented by the user of the Open Nested Service, and be capable of performing the application-specific actions required to compensate for a previously committed transaction

A.2.6 The Implementor's View

The Activity Service defined in the earlier part of this specification enables the development of an advanced transaction model by the definition of appropriate SignalSets, Signals and Actions for that transaction model. This section describes how these entities are defined to implement the Open Nested Transaction Model.

FIGURE 30. illustrates the relationship between an activity which may define a compensating action and the provider of the Open Nested Transaction (ONT) model.

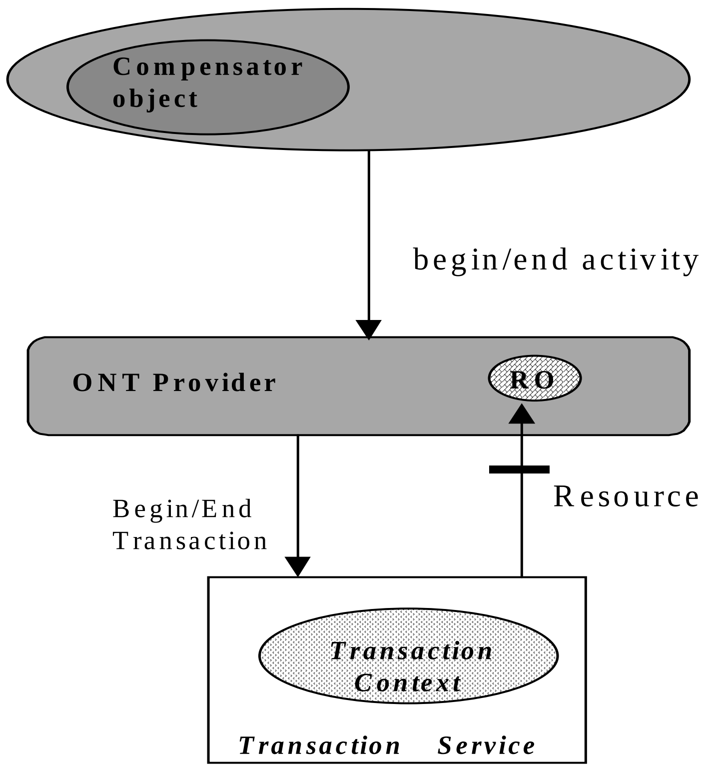

FIGURE 30. Activity and ONT relationship.

Since the Open Nested service provider is responsible for coordinating nested activities according to their final outcomes, it must have the knowledge of the outcomes of their associated transactions in the case of failure. To this aim, the ONT provider participates in the completion of the top level transaction associated with the Activity. That is, it registers as a *javax.transaction.xa.XAResource* object with the transaction. Registering a resource is a way for the ONT provider to determine if the reference of the Compensator object must be logged to be retrieved in case of failure.

Once a Sub-activity has committed, its **Compensator** object must be reachable in case an ancestor Activity rolls back. The ONT provider and the underlying Activity Service Provider are responsible for maintaining access to **Compensator** objects. The path maintained to reach compensating actions is referred to as the *Compensating tree*.

A.2.7 The ONT Service Manager

The Open Nested Service implements the javax.activity.ServiceManager interface with an object that provides the following information to the Activity service:

getServiceInformation returns a ServiceInformation class that has been constructed with a serviceName of “javax.activity.opennested” and a null contextGroup.

getPropertyGroupNames returns null as PropertyGroups are not used my this service.

getCompletionSignalSetName returns the name “javax.activity.opennested.family_outcome”, and **getSignalSet** is capable of providing an instance of that SignalSet.

A.2.7.4 The SignalSet family_outcome

The SignalSet *family_outcome* contains the signals:

- *activity_rolledback*
- *activity_committed*

A.2.7.5 Defined Outcomes

The following identifiers define the Outcome objects used by the ONT protocol:

- *success_with_parent*
- *parent_has_completed*
- *failure_to_invoke_parent*
- *success_with_compensator*
- *failure_to_invoke_compensator*
- *heuristic_compensate_decision*
- *heuristic_cannot_compensate*

A.2.7.6 Role of the ONT provider

Upon receipt of an **activityBegin**, the Open Nested Service:

- invokes the Activity Service to create an activity
- invokes JTA to create a transaction.

Upon receipt of an **activityCommit**, the Open Nested Service:

- invokes the JTA to commit the associated transaction

If the transaction commits

- if a non-null Compensator object was provided in the **activityCommit**, and the commit is related to a nested activity, the Open Nested Service adds an Action object, responsible for compensation, to the **parent** activity; this Action specifies interest in the *family_outcome* SignalSet. If the adding operation fails, the Open Nested Service invokes the Compensator object to *compensate* the committed transaction using the **compensate** operation.
- the Open Nested Service invokes the Activity Service to complete the activity with the completion status **CompletionStatusSuccess** using the *family_outcome* SignalSet.
- If the Open Nested Service receives the **heuristic_compensate_decision** Outcome on the **UserActivity.complete** operation, it throws for the end-user application **HeuristicCompensateException**.
- A null Outcome returned by the **complete** operation on the UserActivity interface is interpreted as an acknowledgment of the completion with success decision.

If the transaction rolls back

- the Open Nested Service invokes the Activity Service to complete the activity with the completion status **CompletionStatusFail** using the *family_outcome* SignalSet.
- If the Open Nested Service receives the **heuristic_can_not_compensate** Outcome, it throws **HeuristicNoCompensateException** to the end-user application.
- A null Outcome returned by the **complete** operation on the **UserActivity** interface is interpreted as an acknowledgment to the completion with failure decision. The Open Nested Service throws the **ActivityRolledBackException** to the end-user.

Upon receipt of an **activityRollback**, the Open Nested Service:

- invokes the JTA to rollback the associated transaction,
- invokes the Activity Service to complete the activity with the completion status **CompletionStatusFail** using the *family_outcome* SignalSet. Any enclosed transactional activity is marked to rollback.

- If the Open Nested Service receives the **heuristic_can_not_compensate** Outcome, it throws **HeuristicNoCompensateException** to the end-user application.
- A null Outcome returned by the **complete** operation on the **UserActivity** interface is interpreted as an acknowledgment to the completion with failure decision.

A.2.7.7 Role of the family_outcome SignalSet

The Activity service provides the SignalSet with a reference to the ActivityCoordinator using the operation **setActivityCoordinator**, and with the CompletionStatus of the Activity using the operation **setCompletionStatus**, before any signals are requested. The SignalSet obtains a reference to its **parent** ActivityCoordinator, if any, by calling **getParent** on the ActivityCoordinator.

According to the activity **CompletionStatus**, a *family_outcome* SignalSet object provides to the ActivityCoordinator when requested with **getSignal**, either

- the *activity_committed* Signal (with the parent ActivityCoordinator returned by **getExtendedDataValue**, or null if there is no parent), if the completion status is **CompletionStatusSuccess**, or
- the *activity_rolledback* Signal (with null returned by **getExtendedDataValue**) if the completion status is **CompletionStatusFail.**

After providing the signal *activity_committed*, if the family SignalSet related to a nested activity receives an outcome response:

- **parent_has_completed** with **setResponse** indicating that an Action fails to be registered with the parent **ActivityCoordinator** because it has completed, the SignalSet indicates to the **ActivityCoordinator** that a subsequent signal shall be sent to that Action. This next signal is *activity_rolledback*.
- **failure_to_invoke_parent** with **set_response** indicating that an Action fails to be registered with the parent **ActivityCoordinator** due to a transient failure or a communication failure, the SignalSet indicates to the **ActivityCoordinator** that the same signal, *activity_committed* shall be sent to that Action. However, if the SignalSet receives the same Outcome **failure_to_invoke_parent** several times it can decide to issue the *activity_rolledback* signal.

After providing the Signal *activity_committed*, if the *family_outcome* SignalSet related to the top-level activity receives

- the outcome response **failure_to_invoke_compensator** with **setResponse** indicating that an Action fails to invoke forget on the Compensator object due to a transient failure or a communication failure, the SignalSet indicates to the **ActivityCoordinator** that the same signal, *activity_committed* shall be sent to that Action.
- the ActionError Outcome indicating any other failure to invoke an Action with the signal *activity_commited* will cause the *family_outcome* SignalSet to return the outcome **heuristic_compensate_decision** on a call to **getOutcome**.

After providing the signal *activity_rolledback*, if a *family_outcome* SignalSet receives

- the Outcome response **failure_to_invoke_compensator** with **setResponse** indicating that an Action fails to invoke *compensate* on the Compensator object due to a transient failure or a communication failure, the SignalSet indicates to the **ActivityCoordinator** that the same signal, *activity_rolledback* shall be sent to that Action. However, after several retrying, the SignalSet can decide to abandon the compensation. Once requested to obtain the final outcome with **getOutcome**, the *family_outcome* SignalSet will return the outcome **heuristic_can_not_compensate**.
- an Outcome indicating that the **ActivityCoordinator** fails to invoke an Action with the signal *activity_rolledback* due a communication failure exception, it informs the SignalSet. The *family_outcome* SignalSet ignores that Action, there is no additional signal or retrying. After a timeout, or once restarted, the ONT provider can ask the **ActivityCoordinator** to obtain the status of the activity; if the status is **StatusCompleted**, the ONT provider will presume that it has rolled back and it invokes **compensate** on the Compensator object.

A.2.7.8 Role of the Action registered with the family_outcome SignalSet

If the Open Nested service Action receives the *activity_committed* signal with a non-null parent **ActivityCoordinator**, it registers with that parent so that it can be informed about its outcome. The Action has knowledge that it represents an Activity that has, nominally, completed successfully, but which may need to be compensated later. The Action then returns the Outcome **success_with_parent**. If the registration with the parent fails because it has completed, it return the Outcome **parent_has_completed**. If the registration with the parent fails due to a transient or communication failure, it return the Outcome **failure_to_invoke_parent**.

This recursively ends up in having all Compensator objects listed in the “family Actions” and having the family Actions registered to the top-level *family_outcome* signalSet as illustrated in Figure D-1.

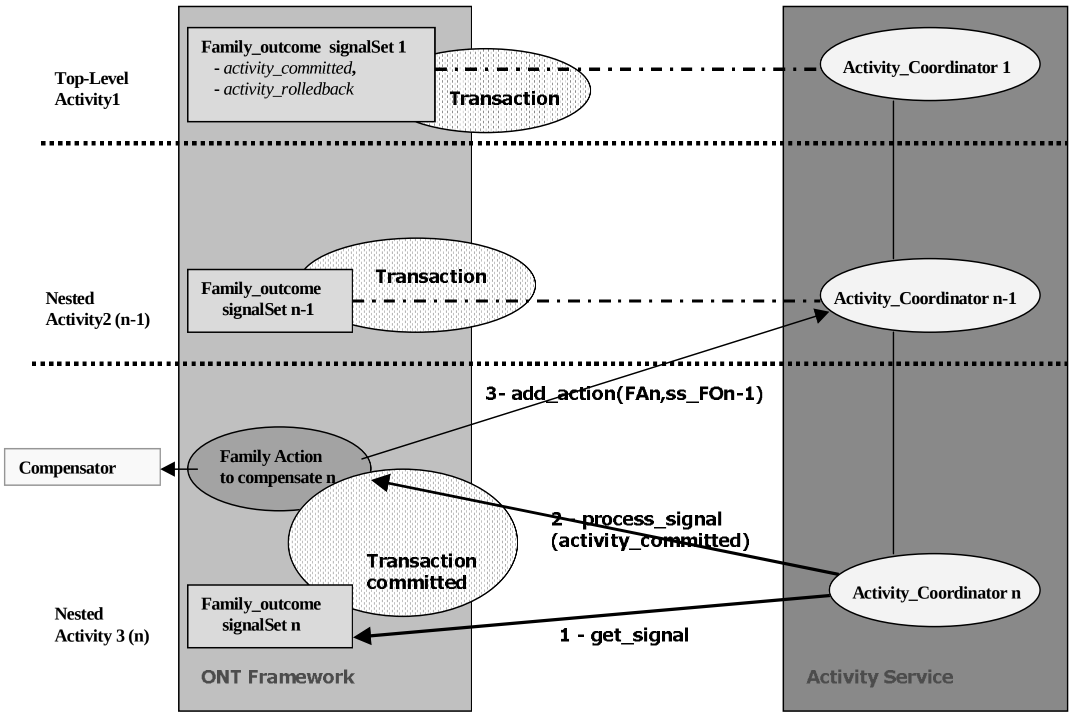

Figure D-1 Transactional Activity commitment and Compensation registrations

If theAction receives the *activity_committed* signal with a null parent **`ActivityCoordinator`**, it invokes a **`forget`** operation on the Compensator object to inform it about the final completion. If the Action fails to invoke the Compensator object it returns the outcome **`failure_to_invoke_compensator`**.

If the Action receives the *activity_rollback* signal, it invokes the **`compensate`** operation on the Compensator object as described in Figure D-2. If the Action fails to invoke the Compensator object it returns the outcome **`failure_to_invoke_compensator`**.

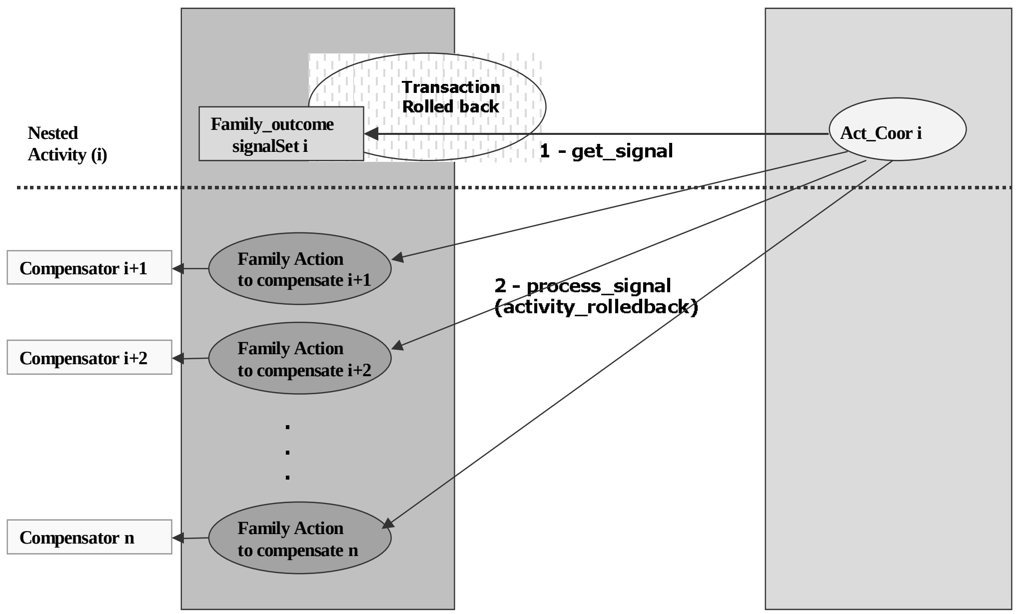

Figure D-2   Compensation on Activity rollback

After a timeout or once restarted, the Open Nested Service has the responsibility to inquire of its associated activity using the **`getStatus`** operation on the **`ActivityCoordinator`**. If the **`ActivityCoordinator`** has completed, it invokes **`compensate`** on the Compensator object. It has the responsibility to retry the **`compensate`** method in case of failure.

An application programmer may invoke the creation of a transaction or an activity within the **`compensate`** operation. However if that transaction rolls back or the activity completes with failure, it is up to application to retry. The Open Nested Service which invokes **`compensate`** is not responsible for its behavior, but only responsible to reach the Compensator object.

A.2.8 Example of Use

The Open Nested service is shown in FIGURE 31. to begin two ONT Activities. The inner activity and its associated transaction complete successfully, so the Compensator for this committed transaction is registered with the outer (parent) activity. In this example, when the parent activity is completed the associated transaction rolls back, so compensation of the first transaction is driven.

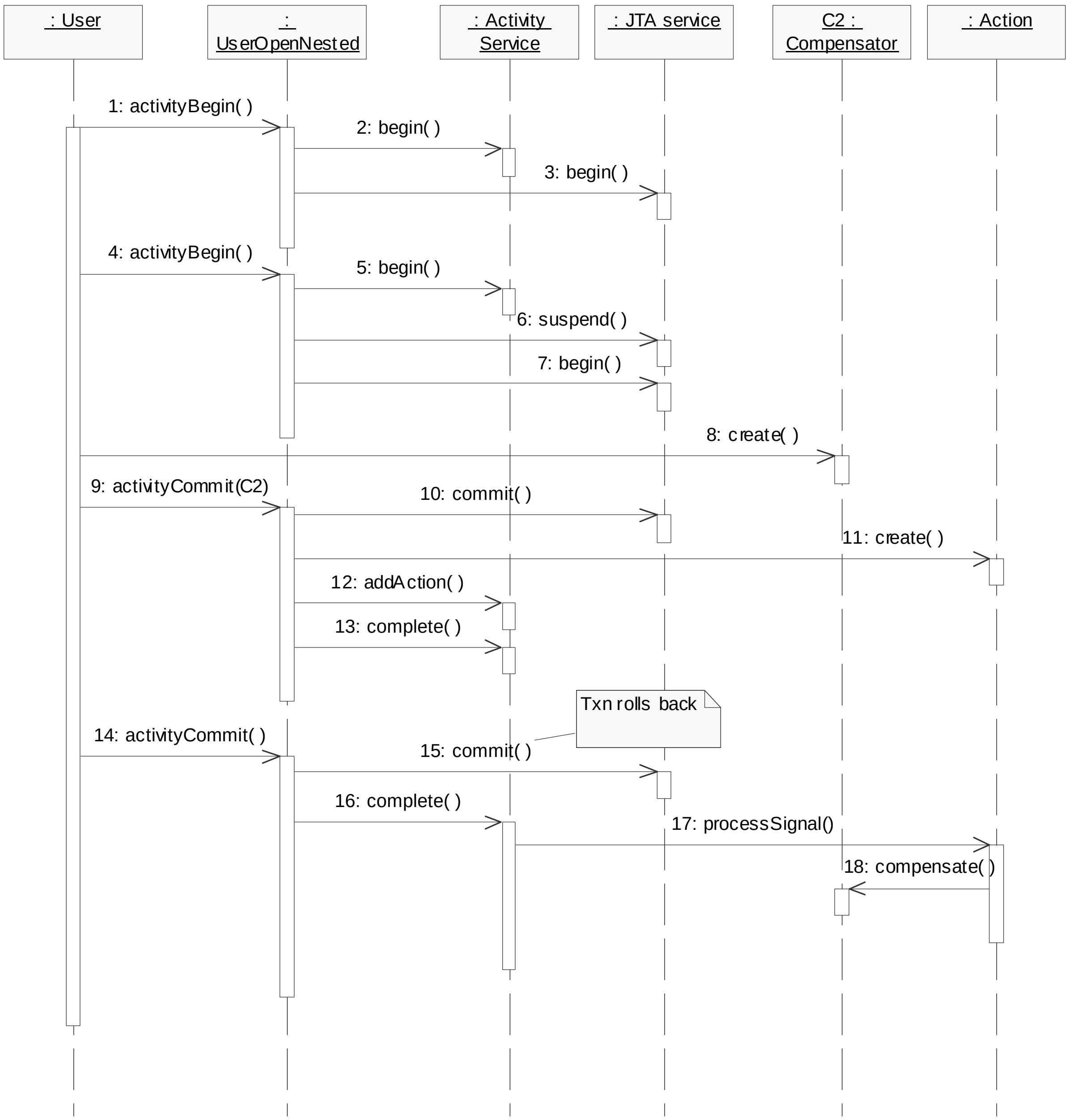

FIGURE 31. Use of Open Nested Service: sequence diagram

1. The user requests that the first (outer) ONT Activity is begun.
2. The Open Nested service calls the Activity service to begin an activity.

3. The Open Nested service calls the transaction service to begin a JTA transaction.
4. The user requets that the second (inner) ONT Activity is begun.
5. The Open Nested Service calls the Activity service to begin a second activity, which is nested by the Activity service within the first activity.
6. The Open Nested Service calls the transaction service to suspend the first JTA transaction and to
7. begin a second JTA transaction.
8. The user performs some transactional work, and creates a Compensator object which can compensate that transactional work if necessary.
9. The user requests that the inner ONT Activity be committed.
10. The Open Nested Service commits the inner transaction successfully.
11. The Open Nested Service creates an Action to represent the Compensator for the committed transaction, and
12. adds it to the parent activity.
13. The inner activity is completed with CompletionStatusSuccess; there are no Actions registered with this activity so no signals are sent.
14. The user requests that the outer ONT Activity be committed.
15. The Open Nested Service tries to commit the associated transaction, but in this example, this transaction rolls back.
16. The Open Nested Service completes the outer activity with CompletionStatusFail, so
17. the *activity_rolledback* signal is sent to the Action which
18. calls **compensate** on the Compensator, thus compensating for the transactional work that was committed in step 10.

To do: rework diagrams to change OMG method names to J2EE names.