Java^{TM}2 Enterprise Edition

J2EE^{TM} Activity Service Specification

JSR095

Specification Lead:
Ian Robinson
IBM Corp.

Technical comments:
Activity service discussion database or
jsr95@hursley.ibm.com

Version 0.2
Expert Group Draft

Trademarks

1.0 Introduction

This document describes the system design and interfaces for the J2EE Activity service. The J2EE Activity service is the realization, within the J2EE programming model, of the OMG Activity service[1].

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 understand *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.

1.1 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.

[1] Additional Structuring Mechanisms for the OTS Specification - *OMG document orbos/2000-06-19*

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.

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[2] 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.

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 package.

1.4 Document Convention

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

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

[2] Java Transaction Service, V1.0, *Sun Microsystems Inc.*

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

1.5 J2EE Activity Service Expert Group

The J2EE Activity service expert group includes the following:

Ian Robinson, ian_robinson@uk.ibm.com
Tony Storey, tony_storey@uk.ibm.com
Tom Freund, tjfreund@uk.ibm.com
IBM (UK) Laboratories
Hursley
Winchester
Hants SO21 2JN
UK

Dave DeCaprio, Daved@alum.mit.edu
i2
5 Cambridge Center
Cambridge, MA 02142

Orest Halustchak, orest.halustchak@autodesk.com
Autodesk Inc.
99 Bank St., Suite 301
Ottawa, ON, K1P 6B9
Canada

Ram Jeyaraman, Ram.Jeyaraman@eng.sun.com
Sun Microsystems Inc,
901 San Antonio Road,
Palo Alto, California 94303,
U.S.A.

Mark Little, Mark@arjuna.com
Bluestone Arjuna Labs,
Churchill House,
12 Mosley Street,
Newcastle upon Tyne,
NE1 1DE,
England.

Ramesh Loganathan, rameshl@pramati.com
Pramati Technologies
301 White House
Begumpet
Hyderabad-500016
India

Eric Newcomer, eric.newcomer@iona.com
IONA Technologies
200 West Street
Waltham, MA 02451 USA

Phil Quinlan, philip.quinlan@bankofamerica.com
Bank of America,
2001 Clayton Road,
Concord, CA, 6420-2405,
USA

Satish Viswanathan, satish@iplanet.com
iPlanet
4850, Network Circle
Santa Clara, CA

USA

Martin West, Martin.west@spirit-soft.com
Spiritsoft
1 Royal Exchange Avenue
Threadneedle Street
London EC3V 3LT

Mehmet C. Eliyesil, jsr000095@silverstream.com
SilverStream Software, Inc.
Application Server Division
Two Federal St.
Billerica, MA 01821

Alex Boisvert , boisvert@intalio.com
Intalio Inc.
2000, Alameda de las Pulgas
suite 250
San Mateo, CA 94403

1.6 Acknowledgements

Alex Mulholland, of IBM, has provided invaluable technical contributions to this specification.

2.0 Overview

The OMG Activity service document[1] describes how an application activity, A_0, may be split into many different, coordinated, short-duration activities which together form a logical long-running business transaction. 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.

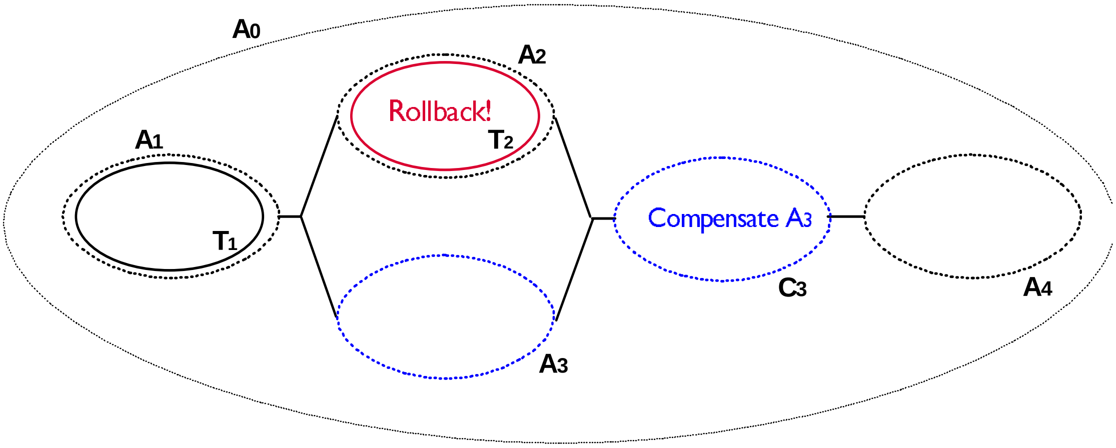


FIGURE 1 **A long-running application activity**

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. For example, in Figure 1 assume that the JTA transaction T_2 encapsulated by A_2 has failed (rolls back). Further assume that the application can continue to make forward progress, but in order to do so must now undo some state changes made in A_1 and A_3. Since T_2 is a transaction, its state changes will be undone automatically by the JTS, so no form of compensation is required for it. Work performed under T_1 has been committed, however, and work performed under A_3 is non-transactional and in this example both need to be compensated. Therefore, new activity C_3 is started as a compensation activity that will attempt to undo state changes performed by A_1 and $A_{3.}$ Once C_3 is complete, forward progress continues with A_4.

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 we are only concerned with their control and coordination, leaving the semantics of such Activities to the high-level services.

The OMG Activity service specifications describes how a variety of unit-of-work (UOW) models may be applied over the Activity service, including OTS strict two-phase commit transactions, nested transactions, as well as a variety of different kinds of transactional behavior including long-running transactions similar to Sagas with Compensation, Open Nested Transactions and Workflows.

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 entities (`Actions`) through `Signals` which are produced by `SignalSets`. `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.

3.0 J2EE Activity Service Architecture

The architecture for a high-level service providing an extended UOW model and using the facilities of the Activity service is shown in Figure 2.

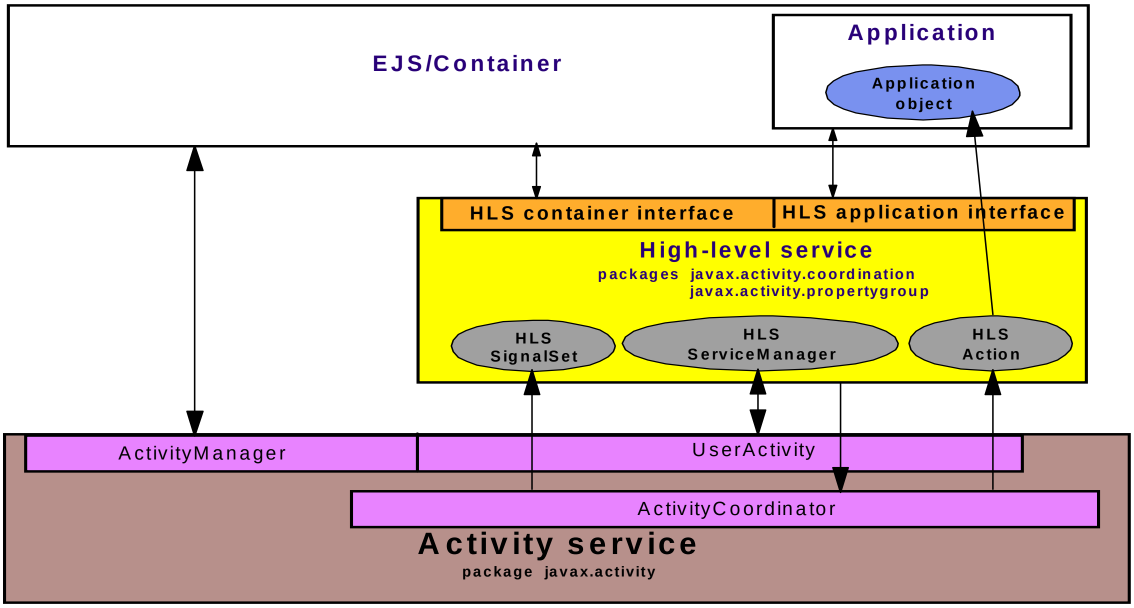


FIGURE 2 **Activity and high-level service architecture**

The architecture is partitioned into three main components:

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 uses the function provided by 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.

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.

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.

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

4.0 Elements of the Activity service

4.1 Features

The features provided by the 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 the `Actions` that have registered an interest. The `ActivityCoordinator` is the dumb messenger of `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 3 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`.

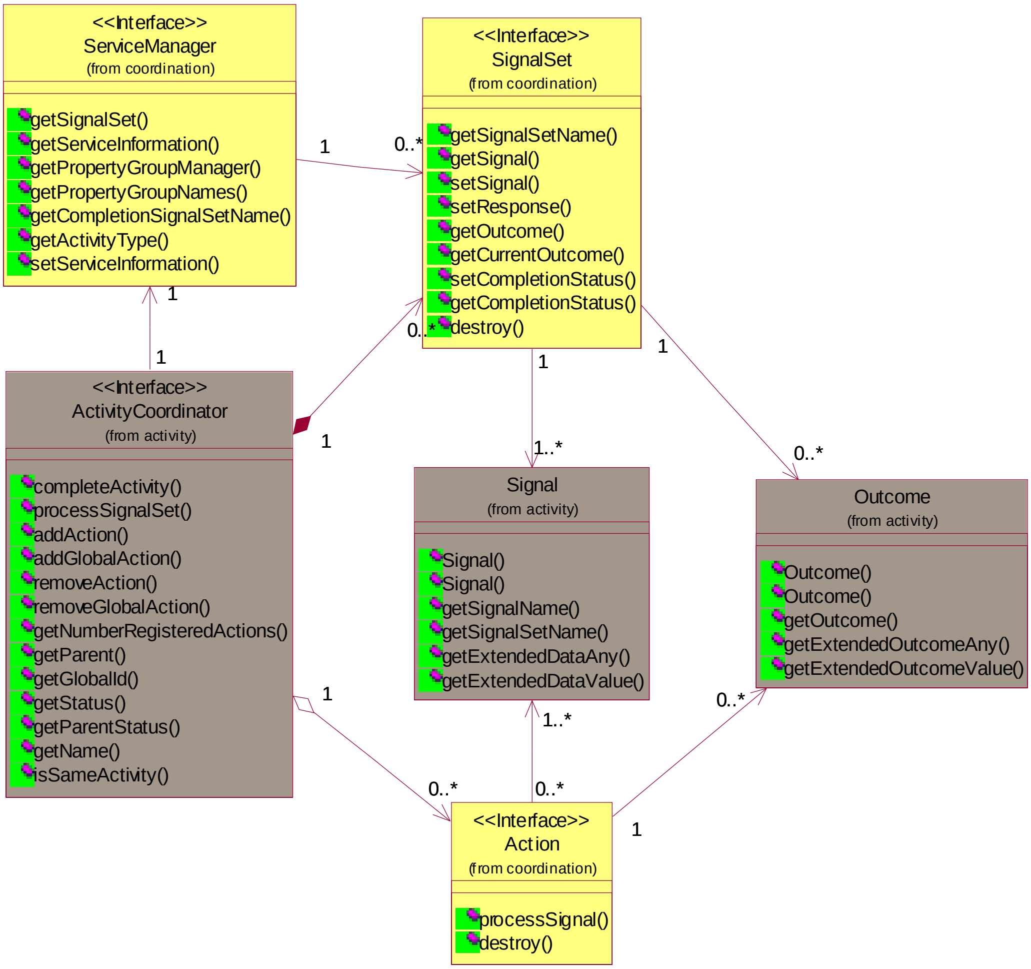


FIGURE 3 **Generic coordination using a pluggable SignalSet provided by the ServiceManager**

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 contexts and (at most one) JTA transaction context may be created in one execution environment 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

`UserActivity` 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 JTS context. This default context group provides nesting behavior compatible with that described in the OMG Activity service.

4.1.3 Distributed service context

The behavior of distributed components running under an Activity context is made location independent by use of an implicit Activity service context that is propagated via IIOP on all remote calls. The Activity service provides 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.

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 relation-

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

ship 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 contains 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 completion `SignalSet` they use.

Each Activity started by a `UserActivity` instance is an Activity instance of the HLS represented by the registered `ServiceManager`. Methods of the `UserActivity` interface that operate on the active Activity context are operating on the HLS Activity instance most recently associated with the calling thread.

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. The ActivityManager interface is a specialization of UserActivity and a HLS should register its ServiceManager with one or the other depending on its requirements. An instance of ActivityManager is obtained via a JNDI lookup of **java:comp/ActivityManager.** ActivityManager instances are only available in an EJB server environment, whereas UserActivity may be made available through a client container.

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.

4.2.1.6 GlobalId

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

4.2.1.7 ActivityCoordinator

The `javax.activity.ActivityCoordinator` is responsible for broadcasting `Signals` to registered `Actions`. It has no logic to understand the `Signals` or the meaning of the resultant `Outcomes`, it simply acts as the messenger. The `ActivityCoordinator` obtains the `Signals`, during broadcasting or completion, from `SignalSets` provided by the HLS.

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.

4.2.1.8 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.

4.2.1.9 Outcome

A `javax.activity.Outcome` is produced by, and given meaning by, an `Action` which has processed a `Signal` or 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.

4.2.1.10 ServiceInformation

An instance of a `javax.activity.ServiceInformation` object is used by each `javax.activity.coordination.ServiceManager` to identify the name of the `ServiceManager` and the context group to which a particular Activity belongs. This information is propagated as part of the `org.omg.CosActivity.ActivityIdentity` structure of the Activity service context, in the `type_specific_data` field, when the type field of the `ActivityIdentity` indicates a J2EE Activity, as described in “Interoperability” on page 31.

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`. Such `Signals` contain an `org.omg.CosActivity.ActivityInformation` structure encoded in an `org.omg.CORBA.Any`. The `org.omg.CosActivity.ActivityInformation` structure is defined in the OMG Activity service specification.

System SignalSets are described in “Predefined SignalSets” on page 20.

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 marshalling and unmarshalling 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 Signaling

A `javax.activity.Signaling` 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 provided 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 *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.

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 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 and which `Signal` should be sent next.

4.2.2.3 Action

A `javax.activity.coordination.Action` is an entity that is 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` registered 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 SignalSets

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

4.3.1 Synchronization

The *org.omg.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 the CompletionSignalSet if the `CompletionStatus` is *CompletionStatusSuccess.*

postCompletion -- sent after all `Signals` produced by the CompletionSignalSet have been distributed.

4.3.2 ChildBegin

The *org.omg.ChildBegin* `SignalSet` contains the signal `childBegin`, which is sent to interested `Actions` 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.

4.3.3 ChildComplete

The *org.omg.ChildComplete* `SignalSet` contains the signal `childComplete`, which is sent to interested `Actions` when a child Activity context has completed. This `Signal` is sent after the *completion* `SignalSet` has finished producing signals but before the *Synchronization* `postCompletion` signal has been processed. The child Activity context is active on the thread when this signal is processed.

4.4 Object Interactions

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

4.4.1 HLS initialization

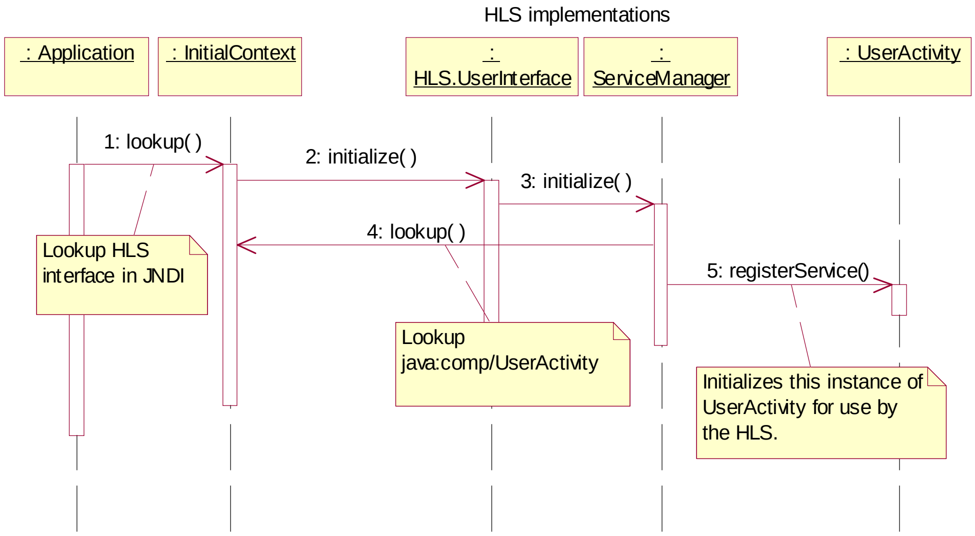


FIGURE 4 HLS initialization sequence diagram.

1. An application performs a JNDI lookup of the application interface provided by a HLS it uses.
2. An instance of the HLS UserInterface object bound in JNDI is created and initialized.
3. The HLS UserInterface object initialization obtains a reference to its `javax.activity.coordination.ServiceManager` interface.
4. The `ServiceManager` performs a JNDI lookup of j**`ava:comp/UserActivity`** to obtain a `UserActivity` instance.
5. The `ServiceManager` initializes the `UserActivity` instance so obtained by registering itself via `registerService`.

4.4.2 Begin an Activity

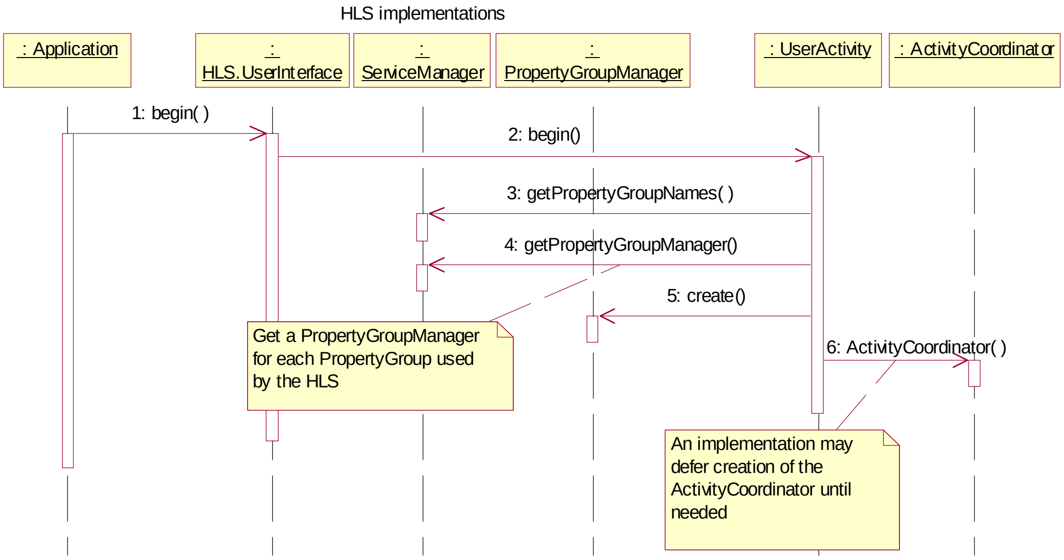

HLS implementations
: Application
: HLS.UserInterface
: ServiceManager
: PropertyGroupManager
: UserActivity
: ActivityCoordinator
1: begin( )
2: begin()
3: getPropertyGroupNames( )
4: getPropertyGroupManager()
5: create()
6: ActivityCoordinator( )
Get a PropertyGroupManager for each PropertyGroup used by the HLS
An implementation may defer creation of the ActivityCoordinator until needed

FIGURE 5 **Activity begin - sequence diagram**

1. An application starts a new HLS activity.
2. The HLS begins a new `UserActivity`.
3. `UserActivity` calls the HLS `ServiceManager` to obtain a list of the names of the `PropertyGroups` used by the HLS's Activities. This information is static and is typically cached in the `UserActivity` instance uring the first request to begin an Activity.
4. For each named `PropertyGroup`, the `UserActivity` obtains an instance of a `PropertyGroupManager` from the `ServiceManager`.
5. Each `PropertyGroupManager` is asked to create a `PropertyGroup` instance to be associated with the new Activity.
6. 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.

4.4.3 Add an Action

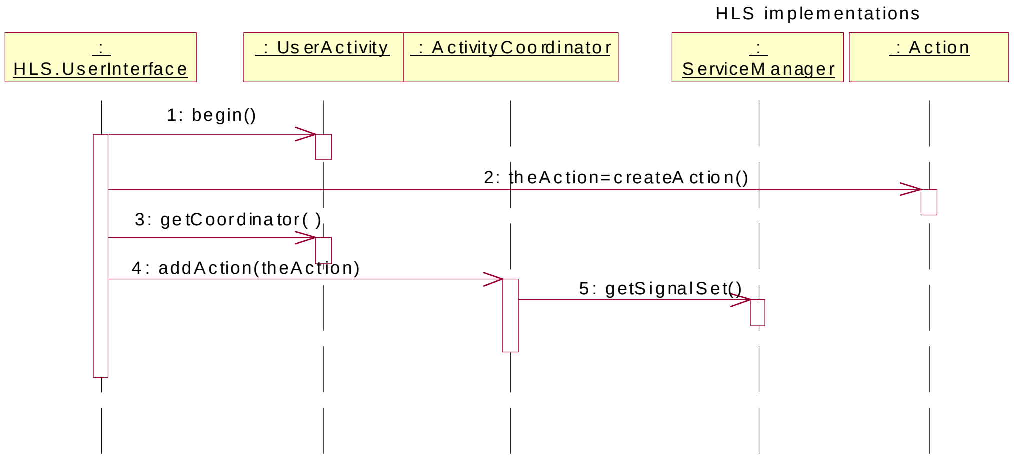


FIGURE 6 **Add an Action - sequence diagram**

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.

4.4.4 Complete an Activity

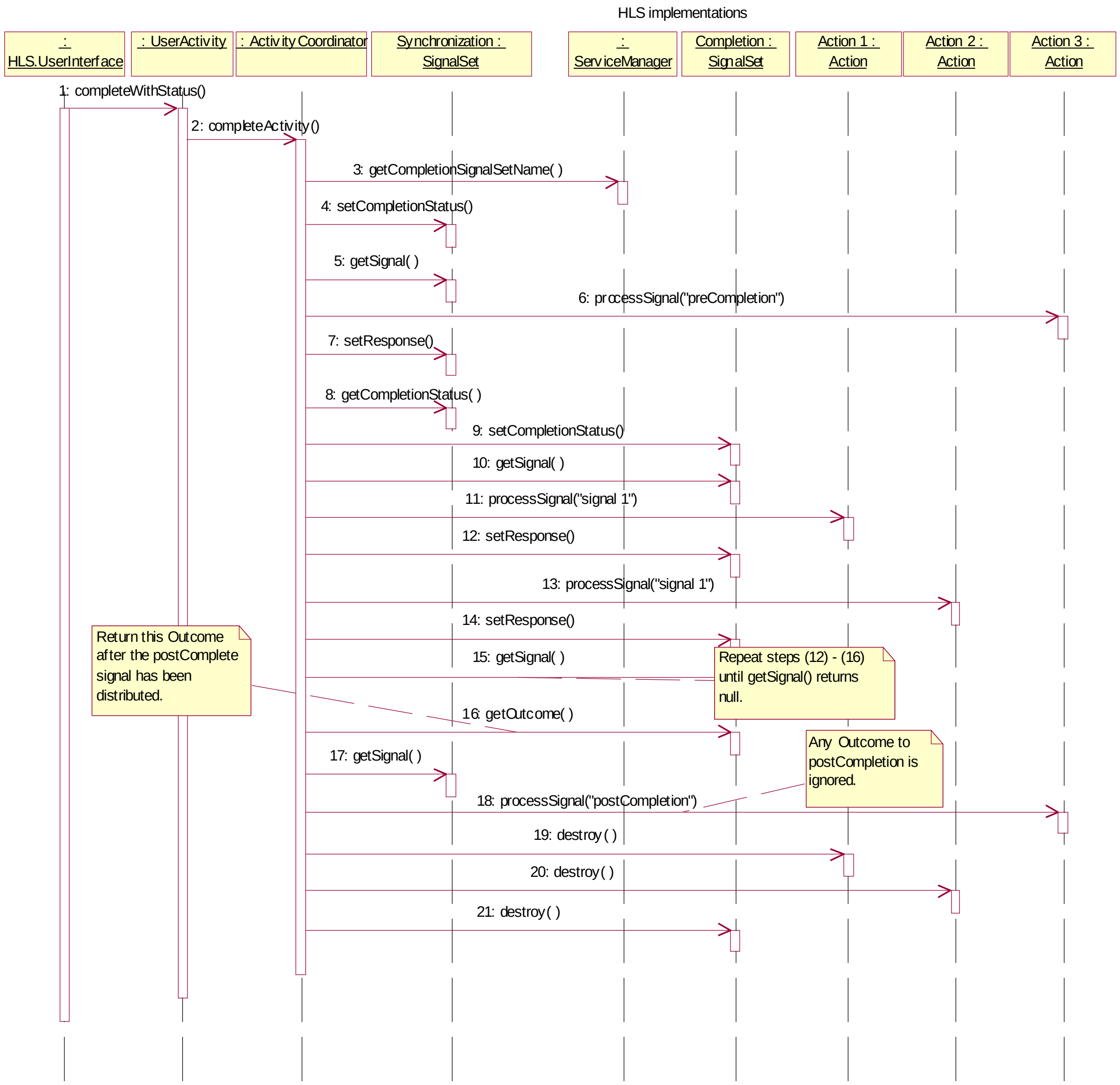


FIGURE 7 Activity completion - sequence diagram

1. An HLS peforms `completeWithStatus`, passing a `CompletionStatus`, for example `CompletionStatusSuccess`.
2. The `UserActivity` object instructs the `ActivityCoordinator` to complete the Activity.

3. The `ActivityCoordinator` obtains the name of the completion `SignalSet` from the HLS `ServiceManager`; 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.
7. This response is passed to the `SignalSet`; the `SignalSet` decides what to do next based on this response and returns a `Signaling` object. The `Signaling` 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`.
12. The `ActivityCoordinator` passes this `Outcome` to the `SignalSet` which factors this `Outcome` into its state table and returns a `Signaling` object that indicates whether to continue sending the current `Signal` and whether to continue involving the current `Action`.
13. Assuming the `Signaling` 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 `Signaling` object returned.
15. If the `Signaling` 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. Before that happens, the `ActivityCoordinator` retrieves the `postCompletion` signal from the *Synchronization* `SignalSet`.
18. It 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`.
19. Actions that are not registered with the *Synchronization* `SignalSet` get explicitly told to `destroy` themselves at the end of the Activity.
20. Ditto 19.
21. Finally, the completion `SignalSet` is told to `destroy` itself. After this, the `Outcome` produced in (16) is returned to the caller.

4.4.5 Broadcast Signals from a SignalSet

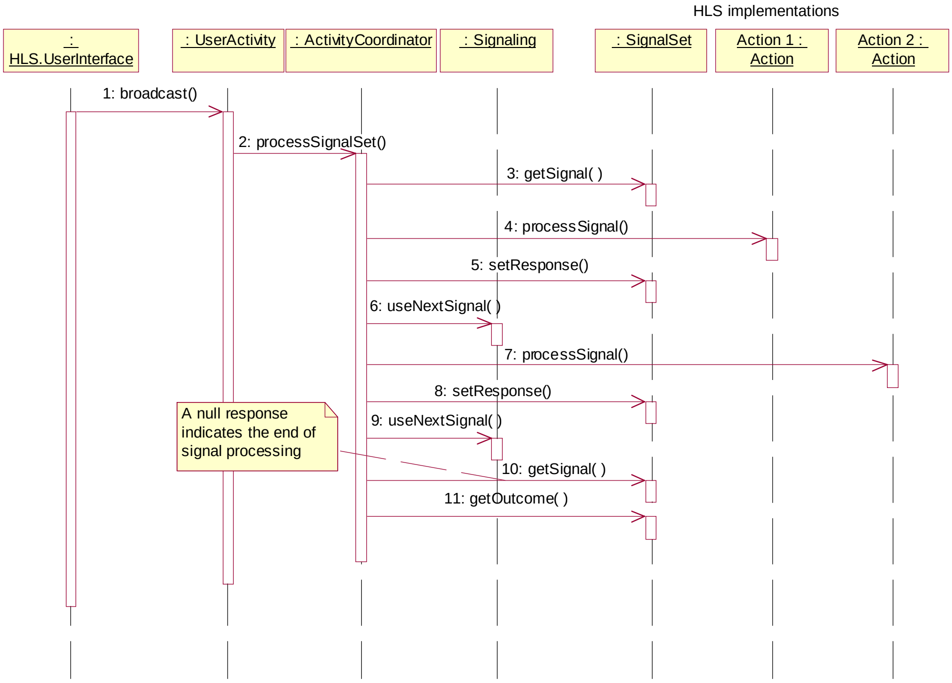
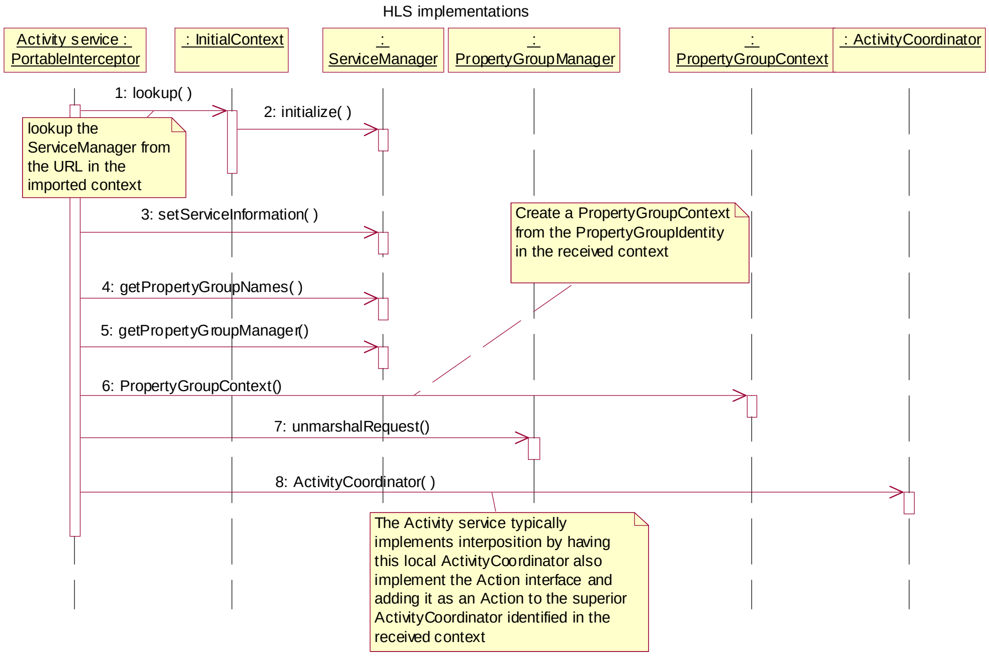


FIGURE 8 Broadcast - sequence diagram

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 drive the `processSignalSet` method of the specific `SignalSet`.
3. The first `Signal` is requested from the `SignalSet`.
4. The `ActivityCoordinator` sends this signal to the highest-priority `Action` interested in the `SignalSet` and obtains an `Outcome` from that `Action`.
5. The `ActivityCoordinator` passes this `Outcome` to the `SignalSet` which factors this `Outcome` into its state table and returns a `Signaling` object
6. The `ActivityCoordinator` enquires of the `Signaling` object whether to continue sending the current `Signal` and whether to continue involving the current `Action`.
7. Assuming the `Signaling` object does not indicate that the current `Signal` should be abandoned, the `Signal` is sent to the next `Action`.
8. Again, the Action’s `Outcome` is fed into the `SignalSet` and a `Signaling` object returned.

9. The `ActivityCoordinator` enquires of the `Signaling` object whether to continue sending the current `Signal` and whether to continue involving the current `Action`.
10. If the `Signaling` 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`.
11. 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.

4.4.6 Import an ActivityContext



HLS implementations
Activity service : PortableInterceptor
: InitialContext
: ServiceManager
: PropertyGroupManager
: PropertyGroupContext
: ActivityCoordinator
1: lookup( )
2: initialize( )
lookup the ServiceManager from the URL in the imported context
3: setServiceInformation( )
Create a PropertyGroupContext from the PropertyGroupIdentity in the received context
4: getPropertyGroupNames( )
5: getPropertyGroupManager()
6: PropertyGroupContext()
7: unmarshalRequest()
8: ActivityCoordinator( )
The Activity service typically implements interposition by having this local ActivityCoordinator also implement the Action interface and adding it as an Action to the superior ActivityCoordinator identified in the received context

FIGURE 9 **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 `type_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 `type_specific_data` is passed to the `ServiceManager`.

4. The interceptor retrieves the list of `PropertyGroup` names supported by the `ServiceManager`.
5. The inceptor 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 interecptor 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.

4.4.7 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.

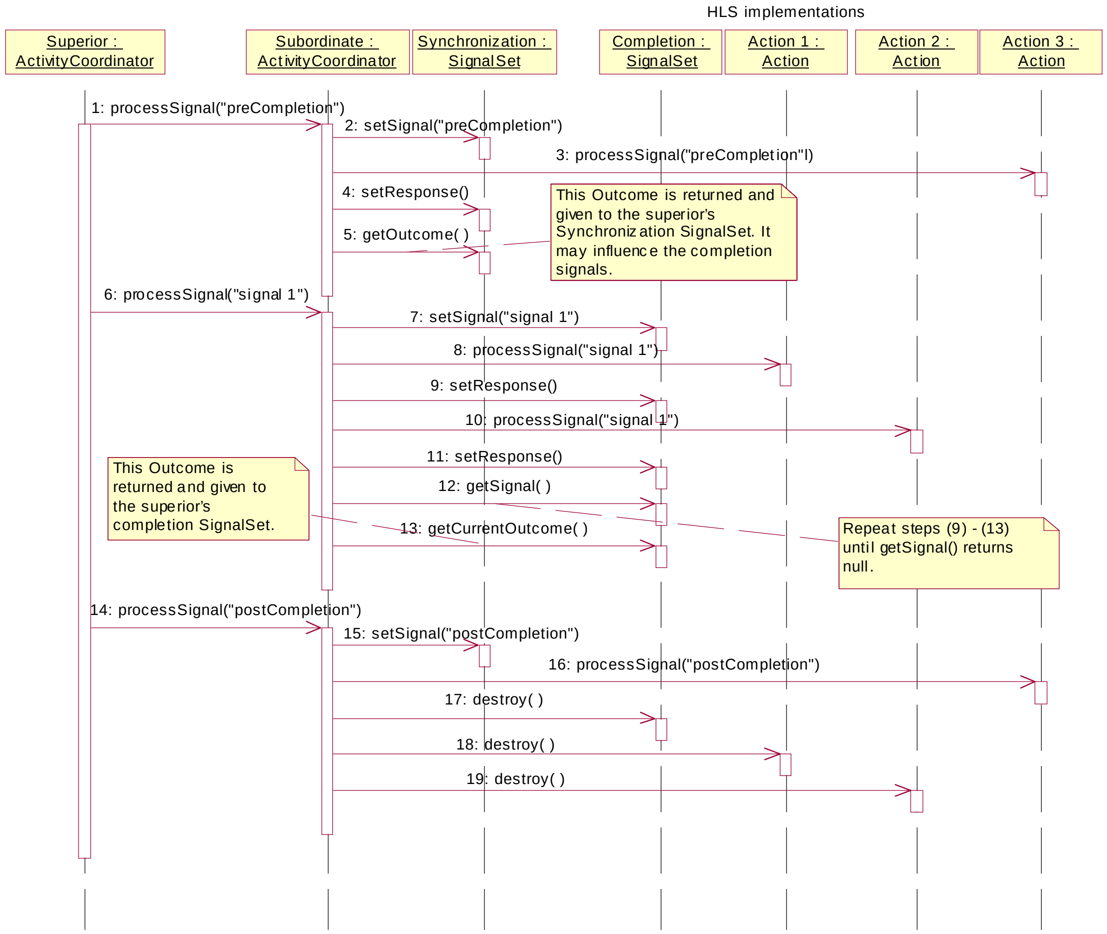

HLS implementations
Superior : ActivityCoordinator
Subordinate : ActivityCoordinator
Synchronization : SignalSet
Completion : SignalSet
Action 1 : Action
Action 2 : Action
Action 3 : Action
1: processSignal("preCompletion")
2: setSignal("preCompletion")
3: processSignal("preCompletion"I)
4: setResponse()
5: getOutcome( )
This Outcome is returned and given to the superior's Synchronization SignalSet. It may influence the completion signals.
6: processSignal("signal 1")
7: setSignal("signal 1")
8: processSignal("signal 1")
9: setResponse()
10: processSignal("signal 1")
11: setResponse()
12: getSignal( )
13: getCurrentOutcome( )
This Outcome is returned and given to the superior's completion SignalSet.
Repeat steps (9) - (13) until getSignal() returns null.
14: processSignal("postCompletion")
15: setSignal("postCompletion")
16: processSignal("postCompletion")
17: destroy( )
18: destroy( )
19: destroy( )

FIGURE 10 **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 `Signaling` object. The `Signaling` 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 Signaling object. The Signaling 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 Signaling 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 Signaling object returned.
12. If the Signaling 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. Ditto 18.

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 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. Although the `org.omg.CosActivity.ActivityIdentity.type` field could be architected to define each specific HLS as they emerge, a better approach would be to architect a specific type value for all J2EE Activities and use the `type_specific_data` field of the ActivityIdentity proposed in the *Additional Structuring Mechanisms for the OTS* Finalization Task Force (FTF) Issue 4305[5] .This proposes the addition of a `type_specific_data` field, of type `org.omg.CORBA.Any`, to the `ActivityIdentity` structure. For a J2EE Activity, this field should contain a `TypeCode` with a `TCKind` of `_tk_struct` and a value of a `j2ee_type_specific_data` structure, defined in IDL as follows:

```
struct j2ee_service_information
{
   string service_name;
   string context_group;
```

[5] URL for OTS-Additional-Structs FTF issue 4305: http://cgi.omg.org/issues/issue4305.txt

```
      any service_specific_data;
   }
   struct j2ee_type_specific_data
   {
      struct j2ee_service_information;
      any extended_information;
   }
```

The `j2ee_service_information` structure is referenced, within the J2EE domain, by a `javax.activity.ServiceInformation` object.

5.2 CORBA interfaces

The `ActivityContext` structure contains references to the following remoteable `CosActivity` interfaces

- org.omg.CosActivity.ActivityCoordinator
- org.omg.CosActivity.Action

A J2EE Activity service provider must provide an implementation of these interfaces that satisfies the requirements for interoperability stated in 5.1 “Requirements on an Activity service implementation”, on page 31.

5.3 CORBA Exceptions

The OMG CosActivity service defines the following new CORBA System Exceptions, each of which have 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 `type_specific_data` are not recognized, an `InvalidActivityException` is thrown.
- If the `service_name` in the `type_specific_data` is not recognized, then Activity context is resumed into the `context_group` defined within the `type_specific_data` in order that the context nesting hierarchy is preserved on flows to downstream domains. The Activity context is otherwise not available to HLS’s 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 HLS’s 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

RMI/IIOP specification -- support the new exception mappings.

OMG Activity service specification -- An `org.omg.CosActivity.ActivityIdentity.type` needs to be allocated for J2EE Activities. There can be a single `type` for all J2EE Activities with the proposal put forward by the *Additional Structuring Mechanisms for the OTS* FTF Issue 4305[5]. A value of `0x4A324545` is suggested. The format of the `ActivityIdentity.type` is described in 5.0 “Interoperability”, on page 31.