OSID Specification Request: 5
Track: Interface
Status: Accepted

Tom Coppeto
Ingenescus

March 2014

Batch Notifications

Status

This document is a request for a specification change for review.

Summary

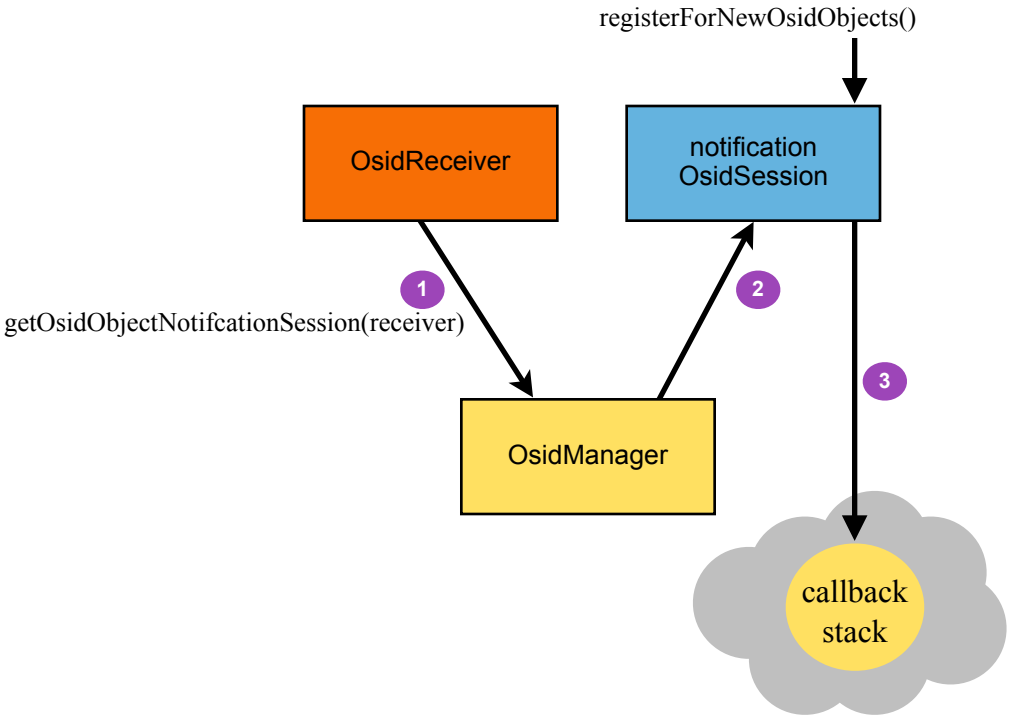
OSID Providers can notify OSID Consumers of events pertaining to the status of OsidObjects through OsidReceiver callback interfaces. OSID Consumers process these events one at a time with no capability for batching events. This document proposes OsidReceiver changes to improve the efficiency of this callback mechanism.

Table of Contents

1. Current Specification....2
2. Problem....4
    2.1. Batch OsidReceiver Callbacks....4
    2.2. Hierarchical Notifications....4
    2.3. Subscription and Notification Alignment....5
3. Proposed Changes....6
    3.1. Typical Bulk OsidReceiver Callbacks....6
    3.2. Hierarchical Notifications....6
    3.3. Atypical Notifications....7
4. Impacts....7
    4.1. Specification....7
    4.2. OSID Consumers....7
    4.3. OSID Providers....8
5. Interoperability Considerations....8
6. Proposed Interfaces....8
    6.1. Example....8
7. Example Scenario....9
8. Copyright Statement....10

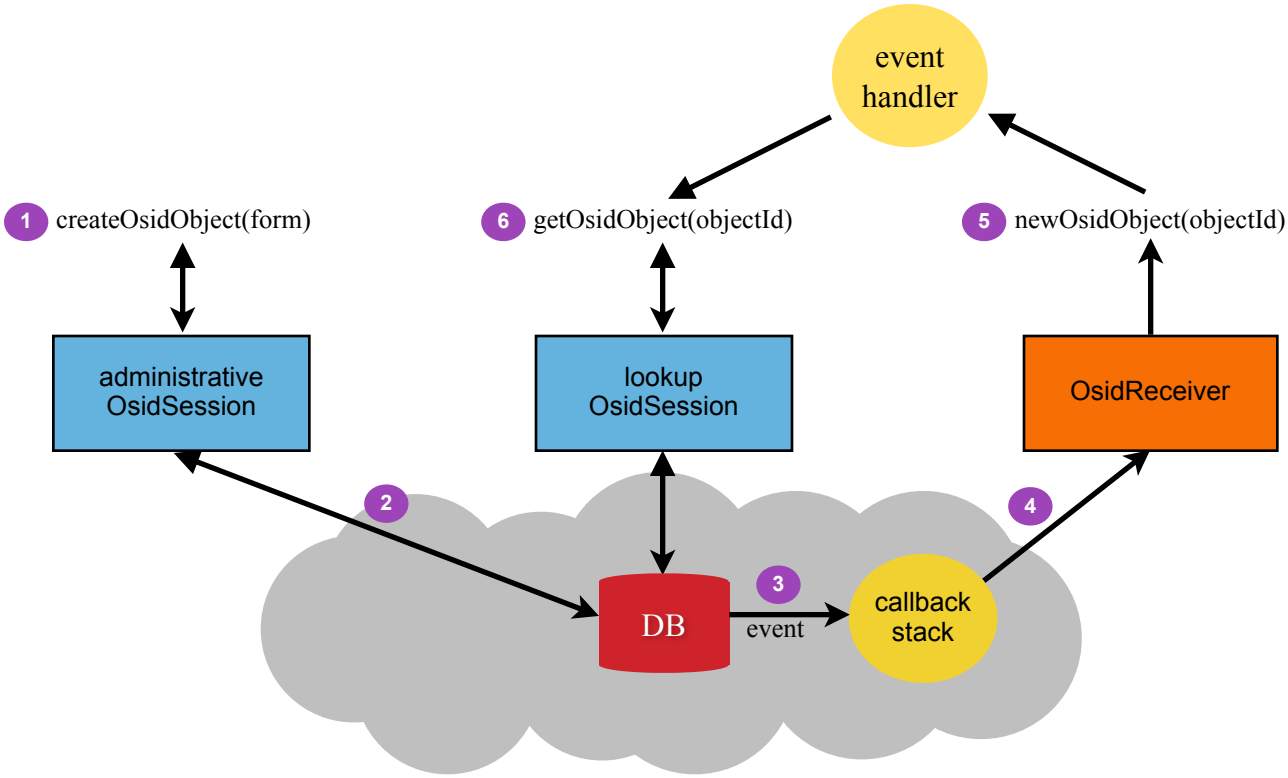
1. Current Specification

OSID Providers can notify OSID Consumers of events pertaining to the status of OsidObjects through OsidReceivers. An OsidReceiver is a consumer-owned object that handles the events. OsidReceivers are supplied when launching a notification OsidSession.



OSID Providers track notification registrations. When an event is triggered, such as the creation of an OsidObject, OSID Consumers who have registered for the event receive a callback via their OsidReceivers.

No information is supplied in the callback other than the Id of the OsidObject. OSID Consumers have to retrieve the OsidObject for the Id supplied in the callback.



Notification sessions and the accompanying OsidReceivers break down OsidObject events along typical create/update/delete boundaries. There are also events for changes in hierarchies.

High level filtering can be performed along the major relations of an OsidObject. Detailed filtering can be performed through the selection of specific OsidCatalogs. The OSIDs do not suggest where this filtering takes place nor what the nature of the message bus is (i.e. multicast or unicast). The callback stack can be maintained anywhere along the path.

The OsidReceiver contains no data other than the identity of the entity to which the event pertains. These notifications are sent outside the context and security perimeter of the event trigger. The recipient may have a different authorization, data presentation, or an entirely separate service endpoint. OsidReceivers are expected to retrieve the OsidObject in response to create and update notifications.

The dilemma for an OSID Provider is to produce a scalable and secure solution among sparse interface touchpoints.

In the simple case, many OSID Consumers listening for an event results a waterfall of retrieval requests. It then becomes the responsibility of the OSID Provider to protect itself by batching events. Considerations for a batch mechanism include throttling and filtering repetitive events.

2. Problem

2.1. Batch OsidReceiver Callbacks

In a batch scenario, the OSID Provider may have a queue of events destined for a service endpoint. The OSID Receiver is limited to notifying the OSID Consumer one at a time.

2.2. Hierarchical Notifications

The bulk callbacks fall short on the existing OsidReceivers. The existing mechanism subscribes to notifications relative to a given node, and new or removed ancestors and descendants of that node are reported.

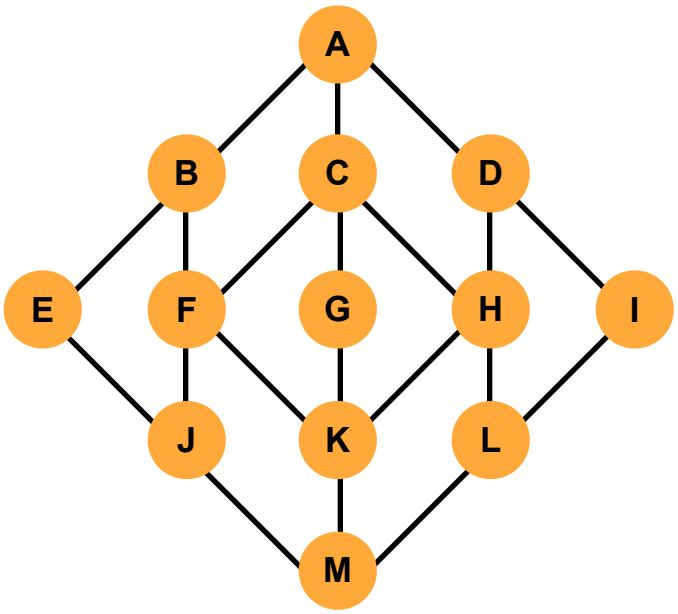
```
newAncestorRepository(repositoryId, ancestorId);
```

If the subscriptions for new ancestors is on node K (diagram below) and node A was added at a later time, then the notification would be:

```
newAncestorRepository(K, A);
```

If C and G were added along with K, then the notification might be (this is a vague interpretation):

```
newAncestorRepository(K, A);
newAncestorRepository(K, C);
newAncestorRepository(K, G);
```



These ancestor & descendant notifications are useful for informing changes to certain hierarchical evaluations, but not enough for tracking the structure of hierarchy. In the above scenario, we don't know what the relations are among A, C & G.

One possibility is to constrain the notifications to parent/child changes.

```
newParentRepository(K, G);
newParentRepository(G, C);
newParentRepository(C, A);
```

In this case, the subscription on ancestors and descendants serve to filter on part of the hierarchy, up or down, but the notifications are on a series of parent/child changes that can be reconstructed on the receiving end. The downside is that specific relationships are sent through the notification bus potentially exposing information based on how specific Ids relate.

Another option is to restrict the notification to what changed without conveying specific relationships.

```
newRepositoryNodes(G, C, A);
```

The receiver would then retrieve the Nodes for these three Ids to determine what in their copy of the hierarchy needs updating (and possibly pushing the need for a bulk node retrieval operation). In this scenario, the receiver maintains the ability to filter notifications above or below a specific node in the hierarchy. These hierarchies serve the purpose of federating so it is most often the case one needs all the nodes from a specific node up to its roots, or all the nodes from a specific node down to its leaves.

2.3. Subscription and Notification Alignment

There exists only a loose alignment between the registration methods in the notification OsidSessions and the OsidReceiver callbacks. For example, two subscriptions routes to the same callback in OsidReceiver:

```
registerForNewEdgesBySourceNode(nodeId);
registerForNewEdgesByDestinationNode(nodeId);
```

```
newEdge(nodeId);
```

Two OsidReceiver implementations, and thus the launch of two notification OsidSessions, would be required if both events needed to be handled separately. Typically, however, it is expected that either event results in a retrieval of the Edge to determine the next course of action.

On create of a Federateable, the OsidObject must be visible in the hierarchy but it has no parents or children. The orphan appears as a root until it is explicitly made a child of another node. The action from moving the node should be to examine the hierarchy to determine its new position. This can be accomplished by examining the immediate nodes around it to determine where it fits in.

If G were inserted:

```
newNode(newNodeId) {
    Node node = hierarchySession.getNodes(newNodeId, 1, 1, true);
}
```

However, if G and C were added simultaneously, two events would fire. An examination of G would reveal a node C that is:

- an orphan because C's hierarchy update has not been received. This would fix itself when C's event arrives.
- unknown because C's create event has not arrived.

Handling asynchronous hierarchy events requires more complexity to maintain proper synchronization and helps to have a reliable and ordered notification bus.

The hierarchical events could be reduced to:

- A new node appeared in the hierarchy that wasn't there before. It was just created and likely an orphan.
- The position of the node changed in the hierarchy because a parent or child was added or removed from the node. This might trigger change events in the surrounding node
- The node was removed from the hierarchy altogether. This would create a hole in the hierarchy that should be "filled in" when change events for the surrounding nodes are received.

The new and removed hierarchy events are also triggered from the new and deleted events on the OsidObject. From an OSID Provider view, these are redundant.

From an OSID Consumer view, it can perform both tasks if it subscribed to both kinds of events. If it was only interested in one or there other event, it would subscribe to one or the other event and gear its OsidReceiver to one or the other case.

The change case differs because change events are triggered for any update that does not effect the hierarchy, and vice versa. A separate change node registration and callback should exist.

3. Proposed Changes

3.1. Typical Bulk OsidReceiver Callbacks

Change the signature of the OsidReceiver callback methods for new, changed, and delete to accept an osid.id.IdList. An IdList, as opposed to an array, aligns well with the getOsidObjectsByIds(idList) lookup method.

3.2. Hierarchical Notifications

In OsidReceiver, replace the ancestor/descendant callbacks with a single change node callback.

```
changedRepositoryNodes(repositoryIds);
```

In notification OsidSessions, replace the ancestor/descendant notifications with:

```
registerForChangedRepositoryHierarchy();
registerForChangedRepositoryHierarchyForAncestors(repositoryId);
registerForChangedRepositoryHierarchyForDescendants(repositoryId);
```

3.3. Atypical Notifications

There are a bunch of random callbacks that do not conform to the bulk pattern. Examples are Resource tracking in osid.mapping, thresholds in osid.metering, linked issues in osid.tracking, in state changes in osid.process. Some (or all) of these are due to the lack of an entity on which to hang the notification. Many of these pattern deviations will be addressed in fleshing out the reporting patterns.

4. Impacts

4.1. Specification

This is a widespread change across all OsidReceivers. A consideration is the limited existence of notification implementations that may make an interface change at this stage of release candidate acceptable. Otherwise, these bulk methods would appear as separate methods OSID Consumers would need to implement.

4.2. OSID Consumers

OSID Consumers would be required to process an osid.id.Id list instead of operating on one osid.id.Id at a time. The osid.id.IdList can be fed directly to a corresponding retrieval method.

Before

```
OsidObjectReceiver {
    newOsidObject(osid.id.Id objectId) {
        OsidObject object = lookupSession.getOsidObject(objectId);
        …
    }
    …
}
```

After

```
OsidObjectReceiver {
    newOsidObjects(osid.id.IdList objectIds) {
        OsidObjectList objects = lookupSession.getOsidObjects(objectIds);
        …
    }
    …
}
```

Tracking hierarchy changes was never implemented well because there wasn't enough information to track the structure.

4.3. OSID Providers

Impact is limited to wrapping a single Id in an IdList for non-federateable OsidObjects. Hierarchy change notifications are incompatible.

5. Interoperability Considerations

The use of an IdList can create an active association between the notifications and retrievals within the same OSID Provider.

One benefit is the added flexibility for an OSID Provider to perform some or all filtering after the notification has been received. While this requires an agreement on the use of the IdList that may be inappropriate for some OSID Consumers, it may be well suited for OSID Adapters that must to some additional processing and cache refreshing along the way.

6. Proposed Interfaces

6.1. Example

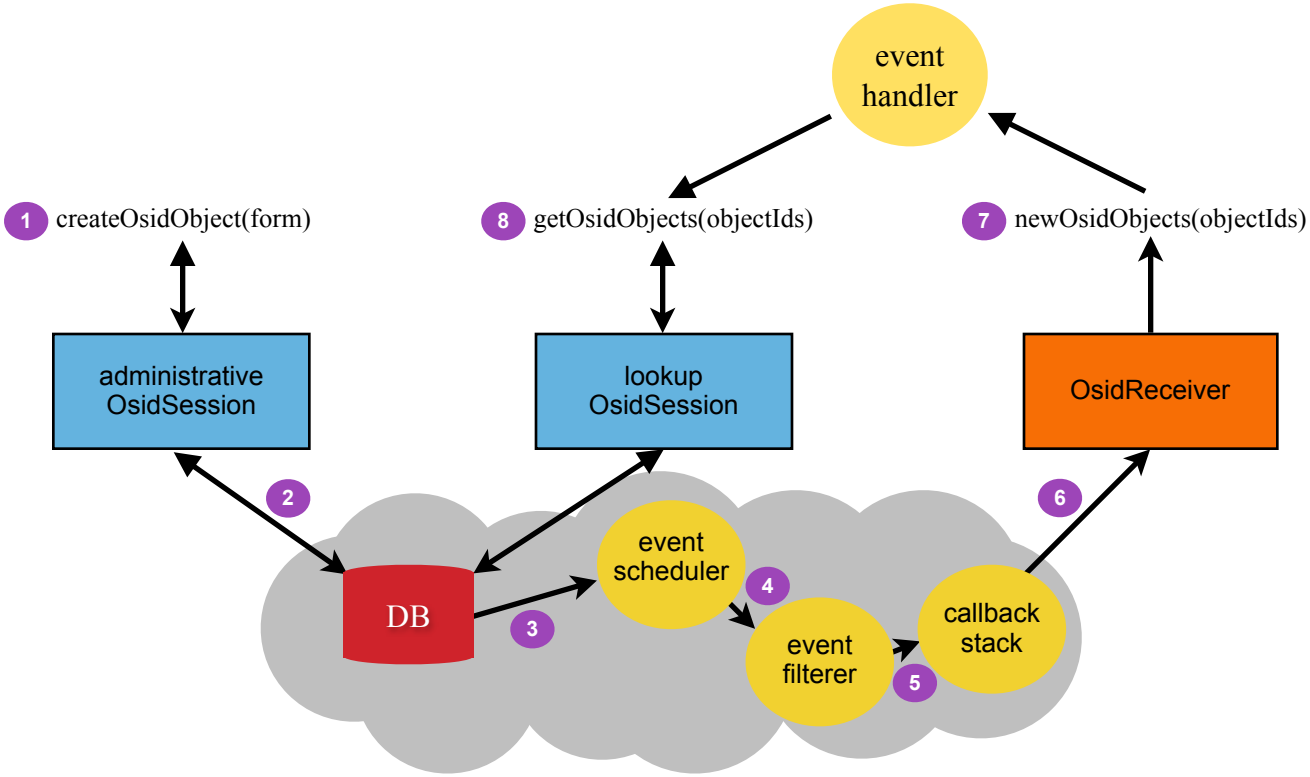
Interface	**osid.assessment.BankReceiver**		
Implements	osid.OsidReceiver		
Description	The bank receiver is the consumer supplied interface for receiving notifications pertaining to new, updated, or deleted Banks.		
Method	**newBanks**		
Description	The callback for notifications of new banks.		
Parameters	osid.id.IdList	bankIds	a list of bank Ids
Compliance	mandatory		This method must be implemented.
Method	**changedBanks**		
Description	The callback for notifications of changed banks.		
Parameters	osid.id.IdList	bankIds	a list of bank Ids
Compliance	mandatory		This method must be implemented.
Method	**deletedBanks**		
Description	The callback for notifications of deleted banks.		
Parameters	osid.id.IdList	bankIds	a list of bank Ids
Compliance	mandatory		This method must be implemented.

Method	changedChildOfBanks		
Description	The callback for notifications of changes to parents and children of bank hierarchy nodes.		
Parameters	osid.id.IdList	bankIds	a list of bank Ids
Compliance	mandatory		This method must be implemented.

7. Example Scenario

An OSID Provider may wish to hold back notifications, batch them at specific times, or filter out repeating operations. In these cases, the OSID Consumer can receive a batch of Ids that are retrieved via a lookup or query OSID Session in bulk.

This allows OSID Providers to shape the incoming traffic in response to system events or other batch operations.



8. Copyright Statement