*Bionic Buffalo Tech Note #50*

# Effects of Portable Object Adapter Policies

*last revised Wednesday 2003.06.18*



## Introduction

The `PortableServer` module from in the CORBA specification defines the `POA` (Portable Object Adapter) interface. When an instance of a `PortableServer::POA` is created, it is associated with certain policies. These policies affect the behaviour of the `POA` and also of other, related objects. This Tech Note summarizes the `POA` policies, how they are used, and the implications for request processing, object identifiers, and other activities and entities.

The information in the Tech Note pertains to CORBA in general and not to any specific CORBA implementation. It is current with the CORBA 3.0. specification (OMG formal/02-11-01). This information is explanatory, and does not include details such as the relevant IDL. For such details, the reader is refered to the specification, especially to the chapter on Portable Object Adapters.

## The `POA` Tree, Object Identifiers and Object References

The various `POA`s are arranged in a tree, beginning at the `RootPOA`. Each `POA` has a name, unique among its siblings, although the children of two different `POA`s might have the same name. `POA` names are null-terminated strings. This scheme allows each `POA` to be identified uniquely by a path starting at the `RootPOA`, similar to the way files are named in a traditional file system by a path from the file system root.

Within a given `POA`, each of the objects belonging to that `POA` is identified by a sequence of octets, the `ObjectId`. The `ObjectId` for an object is unique within that `POA`, so there is a one-to-one correspondence between `ObjectId` and object within the `POA`.

Taken together, these properties imply that each object belonging to an `ORB` may be uniquely identified by knowing that object's `POA` (path from the `RootPOA`) and `ObjectId` (octet sequence).

(One way to implement the key of an `IOR` is to encode the path from the `RootPOA`, and the `ObjectId`, into the key.)

## Policy Objects and POA Creation

Policy objects, which inherit the CORBA::Policy interface, are used throughout CORBA and related specifications. The PortableServer module defines seven specializations of the CORBA::Policy interface. Each of these has a policy value attribute which is an enumeration.

When a POA is created, zero or more of these seven classes of policy objects can be associated with the POA. For each policy object, the associated policy value is associated with the new POA. For each class not explicitly indicated, a default value applies. Once the POA has been created, the associated policies cannot be changed. Note that each POA may use different policy combinations.

In the following tables, the *Min* column indicates whether or not support for the associated value is required by the Minimum CORBA specification. The default value is indicated in the *Interpretation* column.

### ThreadPolicy

The ThreadPolicy specifies the behaviour of threads in POAs. The policy is relevant to multithreaded ORBs. The values are:

| ***ThreadPolicyValue*** | ***Interpretation*** | ***Min*** |
|---|---|---|
| ORB_CTRL_MODEL | *(default)* the ORB assigns requests to threads | yes |
| SINGLE_THREAD_MODEL | no more than one thread may enter any one SINGLE_THREAD_MODEL POA at any given time (multiple requests for any one POA will be queued) | no |
| MAIN_THREAD_MODEL | no more than one thread at a time may enter all MAIN_THREAD_MODEL POAs taken together (multiple requests for any such POA will be queued) | no |

The distinction between SINGLE_THREAD_MODEL and MAIN_THREAD_MODEL is important when a servant makes an invocation which is handled by some other POA's servant. In the SINGLE_THREAD_MODEL, servants of two different POAs might call each other and deadlock.

### LifespanPolicy

The LifespanPolicy value specifies the lifespan of objects implemented by the POA. The values are:

| *LifespanPolicyValue* | *Interpretation* | *Min* |
|---|---|---|
| `TRANSIENT` | *(default)* objects created by the `POA` cannot survive the `POA` itself: once the `POA` is destroyed, the objects must cease to exist | yes |
| `PERSISTENT` | objects created by the `POA` may survive the `POA`: once the `POA` is destroyed, the objects may or may not cease to exist | yes |

The implications of either of these policy values are discussed in more detail, below.

## `IdUniquenessPolicy`

The `IdUniquenessPolicy` value specifies how many `ObjectId` values may be associated with a given servant. The possible values are:

| *IdUniquenessPolicyValue* | *Interpretation* | *Min* |
|---|---|---|
| `UNIQUE_ID` | *(default)* each servant may be associated with no more than a single `ObjectId` value (that is, no more than one object per servant) | yes |
| `MULTIPLE_ID` | each servant may be associated with multiple `ObjectId` values (in other words, there may be more than one object per servant) | yes |

Note that, in either case, there may be times when a servant might be associated with no `ObjectId` values.

## `IdAssignmentPolicy`

The `IdAssignmentPolicy` value determines who assigns the `ObjectId` values: the `ORB`, or the servant.

| *IdAssignmentPolicyValue* | *Interpretation* | *Min* |
|---|---|---|
| `SYSTEM_ID` | *(default)* the `ORB` assigns `ObjectId` values | yes |
| `USER_ID` | the servant assigns `ObjectId` values | yes |

## `ServantRetentionPolicy`

The `ServantRetentionPolicy` value specifies whether or not the `POA` maintains an active

object map. This determines how the POA associates incoming requests (invocations on the methods of specified objects) with servants.

| ***ServantRetentionPolicyValue*** | ***Interpretation*** | ***Min*** |
|---|---|---|
| `RETAIN` | *(default)* the `POA` maintains an active object map, associating `ObjectId` values with servants; the map can be used to associate a given `ObjectId` with a specific servant | yes |
| `NON_RETAIN` | the `POA` does not maintain an active object map; to find the servant for a given `ObjectId`, the `POA` either uses a default servant (`RequestProcessingPolicy = USE_DEFAULT_SERVANT`), or lets a `ServantManager` locate the servant (`RequestProcessingPolicy = USE_SERVANT_MANAGER`) | no |

## `RequestProcessingPolicy`

The RequestProcessingPolicy value specifies how requests are handled by the POA. The possible values are:

| ***RequestProcessingPolicyValue*** | ***Interpretation*** | ***Min*** |
|---|---|---|
| `USE_ACTIVE_OBJECT_MAP_ONLY` | *(default)* the `POA` looks for servants in the active object map | yes |
| `USE_DEFAULT_SERVANT` | the `POA` looks for servants in the active object map (if any); if not found, then the `POA` then forwards the request to the default servant | no |
| `USE_SERVANT_MANAGER` | the `POA` looks for servants in the active object map (if any); if not found, then the `POA` has the `ServantManager` find the servant | no |

These procedures are explained in more detail, below.

## `ImplicitActivationPolicy`

The `ImplicitActivationPolicy` value specifies whether or not implicit activation of servants is supported by the `POA`. Servants may be activated explicitly using several `POA` operations, or (if implicit activation is allowed) they may be implicitly activated. There is a `POA` mapping operation, `servant_to_reference()`, which takes a servant and returns an object reference. In implicit activation, if the servant isn't found in the active object map, the `POA` will automatically create a new object id and reference. This (of course) requires that `IdAssignmentPolicy = SYSTEM_ID` (so

that the system can create the object id) and that `ServantRetentionPolicy = RETAIN` (so there is an active object map in the first place).

| ***ImplicitActivationPolicyValue*** | ***Interpretation*** | ***Min*** |
|---|---|---|
| `NO_IMPLICIT_ACTIVATION` | *(default)* the `POA` does not support implicit activation | yes |
| `IMPLICIT_ACTIVATION` | the `POA` supports implicit activation | no |

## `RootPOA` Policy Values

The `RootPOA` is created during `ORB` initialization, so there is no opportunity to specify policy values for it. Certain values are defined for the `RootPOA` policies. These are:

```
ThreadPolicy = ORB_CTRL_MODEL
LifespanPolicy = TRANSIENT
ObjectIdUniquenessPolicy = UNIQUE_ID
IdAssignmentPolicy = SYSTEM_ID
ServantRetentionPolicy = RETAIN
RequestProcessingPolicy = USE_ACTIVE_OBJECT_MAP_ONLY
ImplicitActivationPolicy = IMPLICIT_ACTIVATION
```

Note that these values are the same as the default values, except for the `ImplicitActivationPolicy`.

Minimum CORBA is not required to support `IMPLICIT_ACTIVATION`, so the values for a Minimum CORBA `ORB` `RootPOA` have `ImplicitActivationPolicy = NO_IMPLICIT_ACTIVATION`, and thus are the same as the default values for a new `POA`.

## Valid Policy Value Combinations

Because of the way the policies affect each other, certain combinations of values are invalid, and certain values require other values. The rules are:

`IMPLICIT_ACTIVATION` requires `SYSTEM_ID` and `RETAIN`

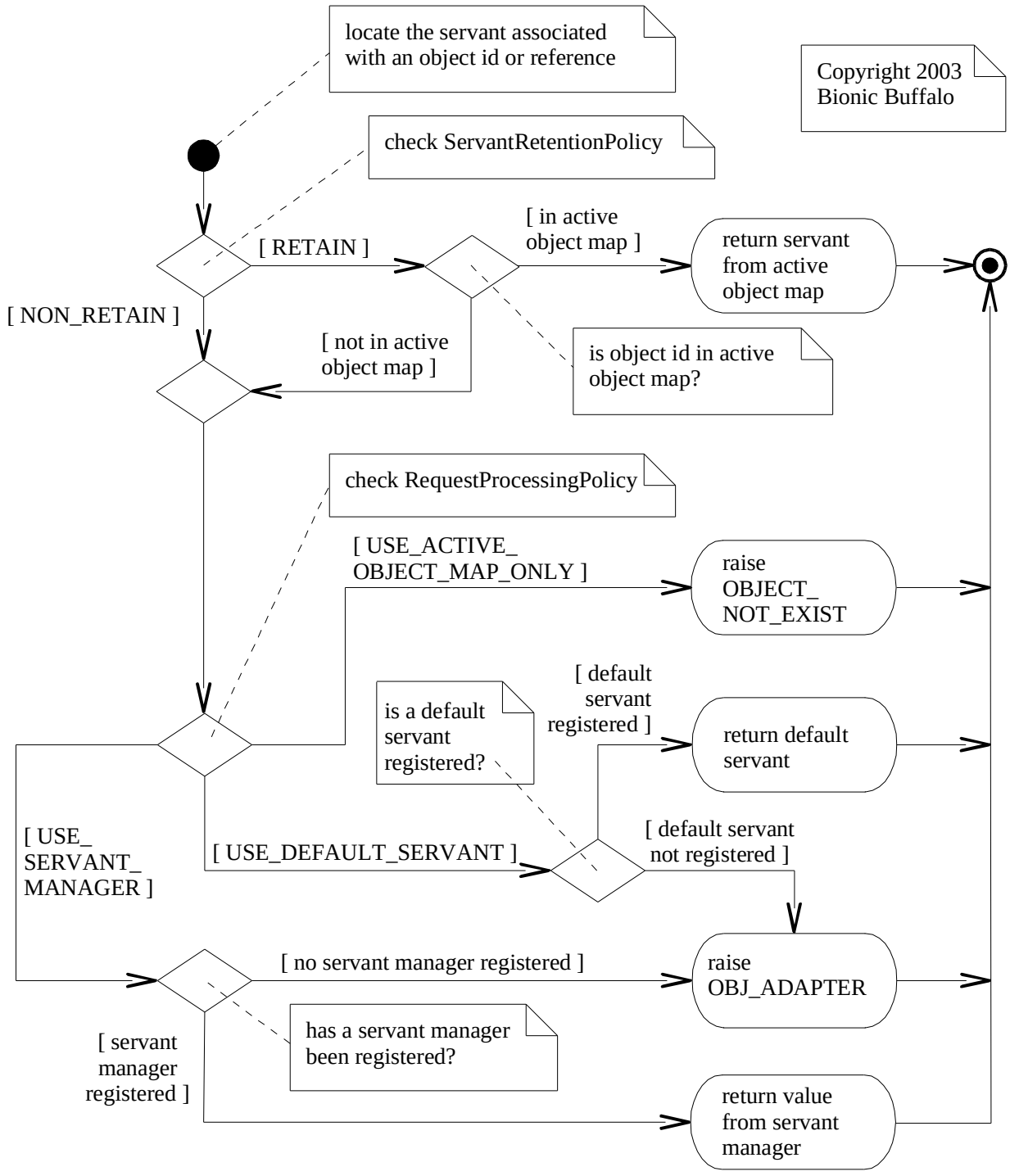
`UNIQUE_ID` is incompatible with `NON_RETAIN`

`NON_RETAIN` requires either `USE_DEFAULT_SERVANT` or `USE_SERVANT_MANAGER`

`USE_ACTIVE_OBJECT_MAP_ONLY` requires `RETAIN`

## Servant Resolution in the POA

Within a POA, servant resolution (finding the servant for an `ObjectId`) consists of the following steps, or of some other set of steps which attains the same end:

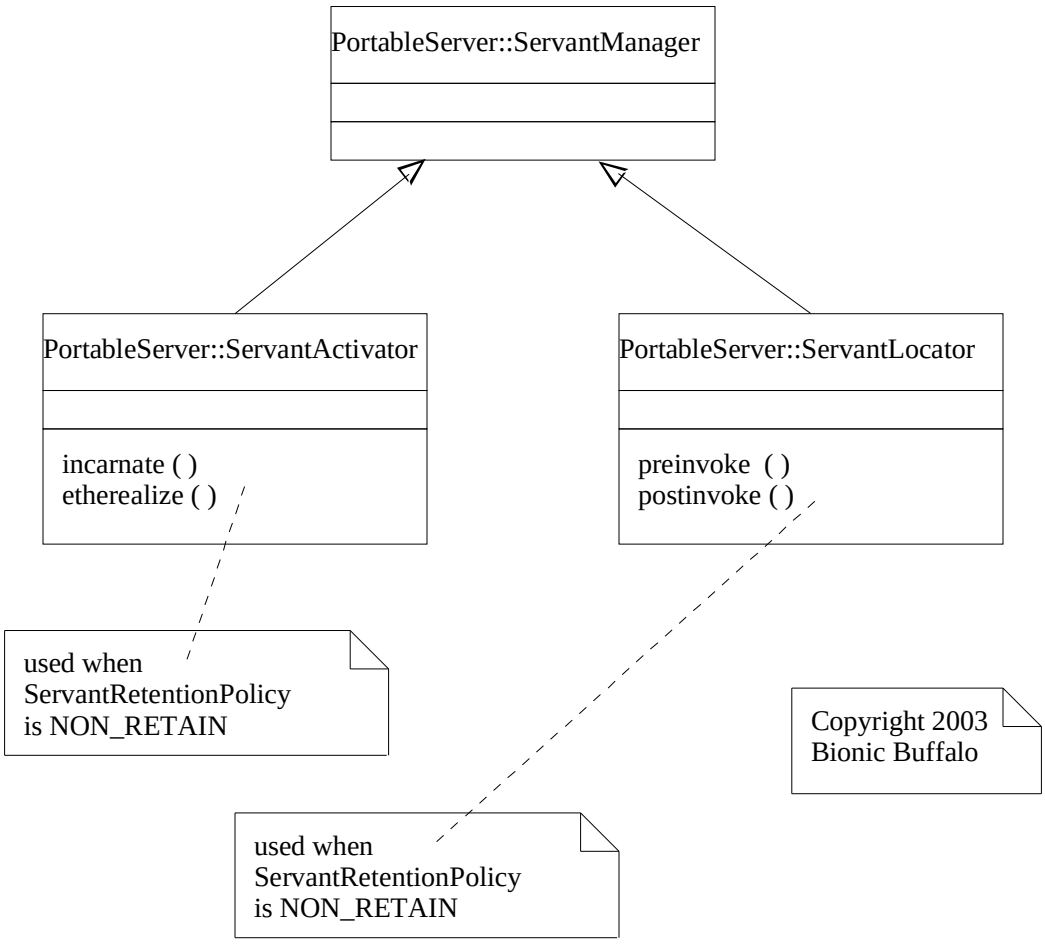


## Varieties of `ServantManager`

When the `RequestProcessingPolicy` is `USE_SERVANT_MANAGER`, then a `ServantManager` is used to get the servant associated with an object reference.

The `ServantManager` itself interface has no interfaces defined. Its purpose is to generalize two other interfaces, `ServantActivator` and `ServantLocator`. When the `ServantRetentionPolicy` is `RETAIN`, then a `ServantActivator` is used as a `ServantManager`. When the policy is `NON_RETAIN`, then a `ServantLocator` is used as a `ServantManager`.

In other words, `ServantActivator` and `ServantLocator` are the two varieties of `ServantManager`.



## Transient and Persistent Objects

The LifespanPolicy value of a POA determines whether or not the objects created by that POA can outlive the POA itself.

Although the POA associated with a persistent object might no longer exist, the object remains associated with that POA. If an invocation is made on a persistent object whose POA doesn't exist, the implementation may elect to create the necessary POA. However, procedures for such automatic resuscitation of a dead POA are beyond the scope of the basic CORBA specification. The Persistent State Service specification does not cover such details.

---



---