Introduction 1

A key feature of many CORBA ORBs is the support of persistent object references. An object reference is persistent if its lifetime can outlive the lifetime of a server for the object. Persistent objects require some kind of activation if no server for the object exists when a client makes an invocation.

Since we support persistent objects with the POA, we already have a server activation framework. However, this framework has some serious deficencies which we will examine here. This document will also describe the existing mechanism we have today as a starting point.

A major motivation for this work is to provide adequate standard APIs in the ORB so that J2EE can be built entirely on the ORB that is supplied with J2SE. This is possible with some small extension to portable interceptors. We will see what extensions are needed and how they can be used in some detail. Note that this is not intended to standardize an ORBD or a particular server activation framework. Instead, we want to provide some standard APIs that allow persistent object references to support a wide range of activation models using only standard APIs.

- proprietary extensions

- implementation repository

Activation Framework 2

2.1 ORBD

2.1.1 Introduction

The ORBD (Object Request Broker Daemon) is used to provide support for the clients to transparently locate and invoke on the persistent objects on servers in the CORBA environment. The persistent servers while publishing the persistent object references in the Naming Service, include the Port Number of the ORBD in the object reference instead of the Port Number of the Server. The inclusion of ORBD port number in the object reference for persistent object references have following advantages:

- The object reference in the naming service remains independent of the server life cycle. For Example, the object reference could be published by the server in the Naming Service, when it is first installed, and then independent of how many times the server is started or shutdown, the ORBD will always return the correct object reference to the invoking client.
- The client needs to lookup the object reference in the Naming Service only once, and can keep re-using this reference independent of the changes introduced due to server life cycle.

The Default Port Number allocated to the ORBD by the Sun's ORB is **1049**. The user or the administrator can always change this default port number by passing the property **com.sun.CORBA.activaton.Port** to the JVM using -D flag.

2.1.2 Design

When ORBD is started up, it creates the following objects:

- **Bootstrap Name Server Object**: The persistent servers publish their object references in this Naming Service. The clients can in turn contact this Naming Service for looking up the object references. The advantage of providing this Bootstrap Naming Service as part of the ORBD is that the user doesn' t need to start an additional Naming Service process for publishing and resolving object references. The Port Number for the Bootstrap Name Server is passed to the ORBD, the client, and server process via property **org.omg.CORBA.ORBInitialPort.**
- **Repository Object:** This object provides interface for the persistent servers to register their Server Definition, e.g., Server Name, Server Program Name, classpath, and various flags or properties that need to be passed into the server process or the JVM when the server is launched. The repository is persistent, i.e.,

the server definition(s) registered with the repository is stored in a file, so that it is available to ORBD in case it goes down and comes back up again. The Repository Object also provides an interface to the external tool, called the **servertool,** to register, unregister, and list server information.

- **Locator Object:** This object is used by the ORBD for: (a). starting up a server if it is not running; (b). locating the listener endpoint associated with a specific ORB in a server; (c). throwing a LocationForwardException with the correct IOR to the invoking client. The Locator Object is also used by the servertool to obtain: (a). a list of endpoints of a specific type associated with all ORBs in a server; (b). all endpoints associated with a specific ORB in a server.
- **Activator Object:** This object is used by the servertool to manage the server lifecycle. For example, the servertool provides commands to: (a). activate/startup a server; (b). shutdown the server; (c). unregister/uninstall the server.

Internally, both activator and locator objects share the same implementation and copy of the ServerManagerImpl. The ORBD creates above objects and publishes them with the Initial Naming Service. The servertool or any other process, e.g., the server process can resolve these references and call appropriate methods to register or obtain the desired information. The ORBD layout is as shown in Figure 1.

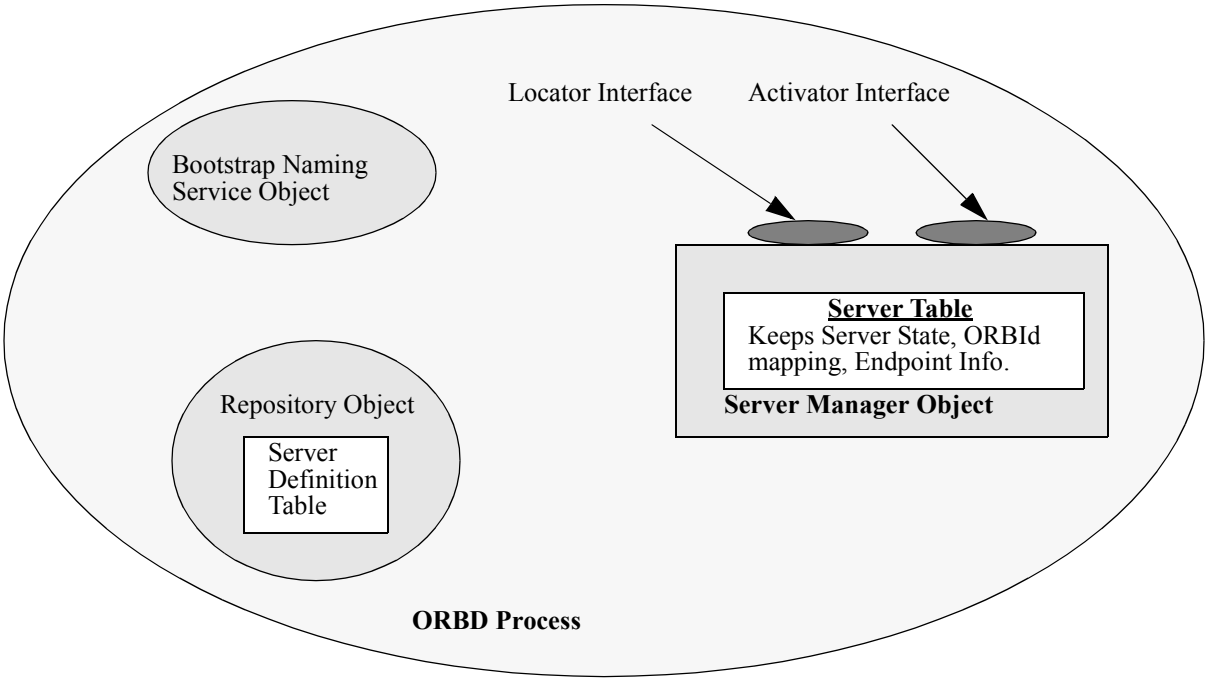


Figure 1: ORBD Composition

2.1.3 ORBD JVM Properties

The ORBD process accepts following JVM properties. Their definition and purpose is explained.

- **com.sun.CORBA.POA.ORBPersistentServerPort:** specifies the listener port for ORBD. The default value for this port is 1049. This port number is added to the port field of the persistent IORs.
- **com.sun.CORBA.Activation.DbDir:** specifies the base where the ORBD persistent storage directory orb.db is created. In the current model, the user.dir system property is retrieved, added to DbDir, and the file orb.db is created in that path.
- **com.sun.CORBA.ORBPersistentServerId:** used to specify the server id to be assigned to this ORBD.
- **org.omg.CORBA.ORBInitialPort:** this property is used to specify the listener for the bootstrap Name server.

2.2 Server Activation

A server can be activated or launched as explained below.

- **Locate:** When a client invokes on the persistent object reference containing the ORBD port-number, if the server is not already running, the ORBD launches or activates the appropriate server.
- **ServerTool:** The user can register a new server definition, or startup an existing server using this external tool.

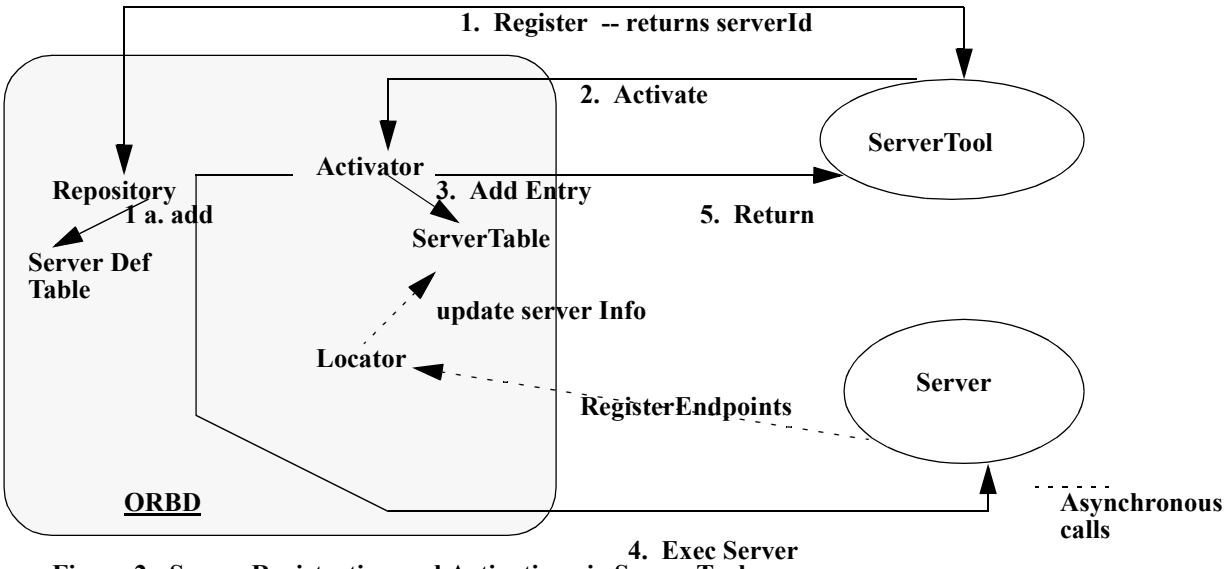


Figure 2: Server Registration and Activation via Server Tool

2.2.1 Server Activation via Server Tool

As shown in Figure 2, the user uses the **register** command in the server tool for registering the server definition with the ORBD. The ServerTool, first calls register() on the Repository to add the server definition to the Server Definition Table. If a new entry is created, a server id is returned to the servertool, otherwise an error of "Already Exists" is returned to the tool. The servertool then calls activate() on the Activator Object. The Activator retrieves the server definition information from the Repository, updates its server table, and launches the server, and returns. The Server once it is activated/launched, calls registerEndpoints() on the Locator Object to register its endpoint(s) and ORB information (ORBName and its mapping) with the ORBD.

Incase, the server has been deactivated, it can be reactivated by using servertool **startup** command. In this case the steps 2 thru 5 are followed from Figure 2. As explained before, the server during its startup process will register its endpoint and ORB information with the ORBD.

2.2.2 Server Activation via Method Invocation

The Server Activation during the method invocation on ORBD is depicted in Figure 3. For this to work, the server definition should have already been registered with the ORBD - Repository. When a client invokes on a persistent object reference, the call is directed to the ORBD. The ORB runtime in ORBD while processing the request, checks the ServerId in the ObjectKey portion of the IOR (Interoperable Object Reference), against the ORBD's ServerId. Since these two ServerIds are different, the **BadServerIdHandler**, registered with the ORBD is invoked. The default **BadServerIdHandler**, looks up the endpoint information corresponding the ServerId and ORBid in the Server Table. If the server is not active, the server is activated, and the information registered by the server is retrieved, and used to form the correct IOR for the client.

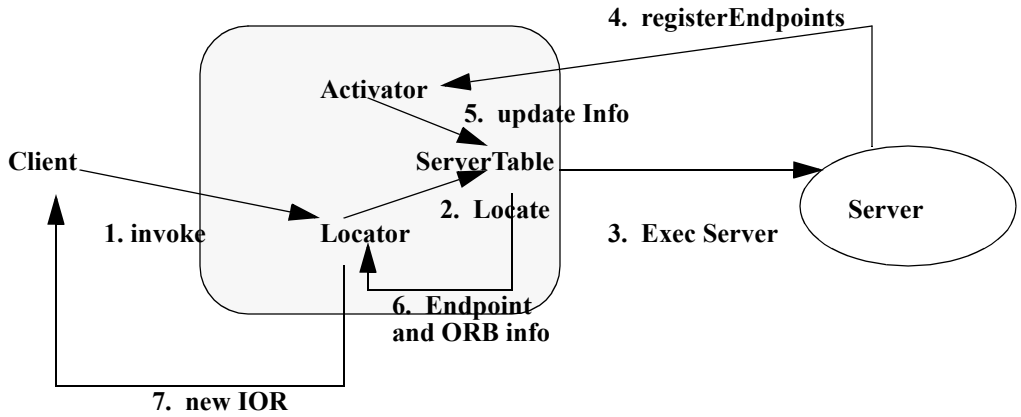


Figure 3: Server Activation during Method Invocation

The application programmers can also register their own BadServerIdHandler class with the ORB using the property **com.sun.CORBA.POA.ORBBadServerIdHandlerClass.**

2.2.3 Server Anatomy

The Application Programmers can structurally organize their server as shown in Figure 4.

```
public class AppServer {

  public static void main() {

     // put code that must be executed during each time a
     // a server is started.  Example is retrieving some
     // information from the database, etc.

            wait();  // for processing Client Requests
}

public static void install() {

    // this piece of code is executed when the server is first
    // registered and activated.  This could be a some code to
    // do some initialization, that needs to be done only once,
    // e.g., creating and publishing Object References into the
   // Namespace.
}

public static void uninstall() {
    // this piece of code is executed when a server is
   // unregistered from the Repository.  This could be some
  // piece of code to cleanup the specific objects or data.

}

public static void shutdown() {
    // this piece of code is executed when the ORB is shutdown.
   // This code could contain logic to call a shutdown on each
   // of the ORBs created by the User Application.
}
```

Figure 4: The Structural organization of the Server Code

The main() method is executed each time a server is activated. Therefore, this section should contain the code that can be executed any number of times. For example, it could retrieve some server state during each activation.

The install() method is executed only during server initialization. For example, this method could contain the code to initialize the ORBs and to create and publish object references in the Namespace. This server method is invoked by the ORB Server Activation Framework, when the server is first registered and activated.

The uninstall() method is executed only when the server is unregistered from the Repository. This method could contain the code to delete Objects installed by this server in the Namespace. For example, when the server definition is deleted from the Repository, the object references created by the deleted server become unusable, and should be cleaned up from the Namespace.

The shutdown() method is called to shutdown the server. The server which has been shutdown can be re-activated again. The typical application programmer implementation of this method will be to shutdown each of the ORBs associated with this server.

Note: There were a few discussions on overriding the shutdown semantics provided by the JDK1.3 and above. The install(), uninstall() and shutdown methods are optional for the Application Server writers. The Application programmers can write the whole server logic in the main method, this way their applications will stay portable across multiple ORBs.

2.3 Server Tool

The Server Tool provides the ease of use interface for the application programmers to register, unregister, startup, and shutdown a server. In addition to above four commands other commands are provided to obtain various statistical information about the server. The Server Tool commands and their brief function is disussed below.

2.3.1 Register Server

This command is used to register a new server defintion with the Implementation Repository. The information passed to this command includes: (a). server class name; (b). server name; (c). classpath to the server class; (d). any arguments to be passed to the server; and (e). any flags to be passed to the Java VM.

If the defintion is not in the Implementation Repository, a new entry is created with the supplied information, and the server is activated or launched. In case of an error, an appropriate exception is thrown.

The syntax for this command is:

register -server <server class name> -applicationName <alternate server name> -classpath <classpath to server> -args <args to server> -vmargs <args to server Java VM>

2.3.2 Unregister Server

This command is used to delete a server's definition from the Implementation Repository. Given the server id or server name (passed during server registration), the server tool: (a). contacts the Activator Object to shutdown the server, and delete its information from its server table; and (b). contacts the Implementation Repository to delete the server definition. The syntax for this command is:

unregister [-serverid <server id> | -applicationName <name>]

where server id is obtained after the server is registered, and applicationName is the alternate server name provided to the registration command.

2.3.3 Locate Server

This command is used to locate the endpoints of a particular type for all ORBs created by the Server. If a server is not already running, then it is activated. The syntax for this command is:

locate [-serverid <server id> | -applicationName <name>] [-endpointType <endpointType>]

If an endpointType is not specified, then the Plain/non-protected endpoint associated with each ORB in a server is returned.

2.3.4 Locate Server For An ORB

This command is used to locate all the endpoints registered by a specific ORB of a Server. If a server is not already running, then it is activated. The syntax for this command is:

locateperorb [-serverid <server id> | -applicationName <name>] [-orbid <ORB name>]

If an orbid is not specified, then the default value of "" is assigned to the orbid. If there are any ORBs created with an orbid of empty string, then all the ports registered by it are returned, otherwise an error message is returned.

2.3.5 ORB Id Mapping

This command is used to list the integer mapping for the ORBIds. The ORBIds are the string name for the ORB created by the Server. When a server initializes an ORB with a particular ORBId, an integer mapping for that particular ORBId is obtained. This integer mapping that is put into the object key, to help in locating the correct ORB in the server during requests on the ORBD. If the server is not already running, then it is activated. The syntax for this command is:

orblist [-serverid <server id> | -applicationName <name>]

2.3.6 Get Server ID

This command is used to retrieve the server id corresponding to the server application name from the Implementation Repository. The syntax for this command is:

getserverid [-applicationName <name>]

2.3.7 List of Servers in the Implementation Repository

This command is used to retrieve information about all servers registered with the ORBD, and whose definition is present in the Implementation Repository. The syntax for the command is:

list

In response to this command, the server id, server name, and the corresponding server application name for each server in the Implementation Repository is retrieved and displayed to the user.

2.3.8 List of Active Servers

This command is used to retrieve the information about all active servers on a machine. The active servers are the one's which have been launched by the Activator and as still running. The syntax for this command is:

listactive

In response to this command, the server id, server name, and the corresponding server application name for each active server is retrieved and displayed to the user.

2.3.9 List Application Names

This command is used to list the application names for all the servers that are currently registered with the ORBD. The syntax for this command is:

listappnames

2.3.10 Start Server

This command is used to startup or activate the server. If the server is not running, this command will launch the server. In case the server is already up and running an error message is returned to the user. The syntax for this command is:

startup [-serverid <server id> | -applicationName <name>]

In case of errors, an appropriate error message is returned to the user.

2.3.11 Shutdown Server

This command is used to shutdown the active server. During the execution of this command, the the shutdown() method defined in the server application program is also invoked to shutdown the server process appropriately. The syntax for this command is:

shutdown [-serverid <server id> | -applicationName <name>]

2.3.12 Help

The help command is used to list all the commands available to the server thru the server tool. The syntax for this command is:

help

2.3.13 Quit

This command is used to quit out of the server tool. The syntax for this commnad is:

quit

Note: The current ServerTool version also provides three commands to list the poa name to poa id mapping, poa id to poa name mapping, and the list all poa names and ids registered so far. In the current architecture, since this mapping is stored as part of the ORBD object, they are available. Eventually, we will eliminate the POAIdMapper Object from ORBD, and move it to the POAORB (TBD by Ken Cavanaugh). When that happens, we need to remove these three commands from the ServerTool.

2.4 Example Code

This section provides an example client and server code that would work with the current **Server Activation Framework**.

2.4.1 Example IDL File

```
module examples {
    interface policy_2
    {
        long increment();
    };
};
```

2.4.2 Example Server Program

```
package examples;

import java.util.Properties;

import org.omg.CORBA.Object;

import org.omg.CORBA.*;

import org.omg.CosNaming.*;

import org.omg.CosNaming.NamingContextHelper;

import org.omg.CORBA.ORBPackage.InvalidName;

import org.omg.PortableServer.POAManagerPackage.AdapterInactive;

import org.omg.PortableServer.POAPackage.InvalidPolicy;

import org.omg.PortableServer.POAPackage.AdapterAlreadyExists;

import org.omg.PortableServer.POAPackage.WrongPolicy;

import org.omg.PortableServer.POAPackage.ServantAlreadyActive;

import org.omg.PortableServer.POAPackage.ServantNotActive;

import org.omg.PortableServer.*;

import org.omg.PortableServer.POAPackage.AdapterAlreadyExists;

import org.omg.PortableServer.POAManagerPackage.AdapterInactive;

import org.omg.PortableServer.IdAssignmentPolicyValue;

import org.omg.PortableServer.ThreadPolicyValue;

import org.omg.PortableServer.LifespanPolicyValue;

import org.omg.PortableServer.IdUniquenessPolicyValue;

import org.omg.PortableServer.ServantRetentionPolicyValue;

import org.omg.PortableServer.RequestProcessingPolicyValue;

import org.omg.PortableServer.ImplicitActivationPolicyValue;

import org.omg.CORBA.Policy;

import org.omg.PortableServer.Servant;


class policy2_servantA extends policy_2POA

{
```

```
    private int countValue;
    public policy2_servantA()
    {
        countValue = 0;
    }
    /**
     * Implementation of the servant object.
     * The funtion intakes no parameter
     * and returns an int value incremented by one.
     */
    public int increment()
    {
        return ++countValue;
    }
}

class policy2_servantB extends policy_2POA
{
    private int countValue;
    public policy2_servantB()
    {
        countValue = 1000;
    }
    /**
      * Implementation of the servant object.
      * The funtion intakes no parameter
      * and returns an int value incremented by one.
      */
```

```
    public int increment()
    {
        return ++countValue;
    }
}
public class policy2Server
{
    private static policy2_servantA acs1;
    private static policy2_servantB acs2;
    private static org.omg.CORBA.ORB orb1;
    private static org.omg.CORBA.ORB orb2;
    private static org.omg.CORBA.Object obj1;
    private static org.omg.CORBA.Object obj2;
    private static Integer initialized;
    static {
        acs1 = new policy2_servantA();
        acs2 = new policy2_servantB();
        initialized = new Integer(0);
    }
    private static final String msgPassed = "policy_2: **PASSED**";
    private static final String msgFailed = "policy_2: **FAILED**";
    public static void main( String args[] )
    {
        try
        {
            initializeORBs();
            // publish objects in the Namespace
            publishObjects(orb1, obj1, "Object1");
            publishObjects(orb2, obj2, "Object2");
            System.out.println( "Policy_2 Server is Ready and Waiting" );
```

```
            java.lang.Object sync = new java.lang.Object();
            synchronized( sync )
            {
                sync.wait();
            }
        } catch( Exception exp ) {
        exp.printStackTrace();
        System.out.println( msgFailed + "\n" );
    }
}
private static void initializeORBs() {
    try {
         if (initialized.intValue() == 0) {
            orb1 = initializeORB("suborb1");
            orb2 = initializeORB("suborb2");
            //create the rootPOA and activate it, and publish objects in Namespace
             obj1 = activatePOAs(orb1, acs1);
             obj2 = activatePOAs(orb2, acs2);
             initialized = new Integer(1);
         }
    } catch(Exception exp) {
        exp.printStackTrace();
    }
}
public static void publishObjects(org.omg.CORBA.ORB orb, org.omg.CORBA.Object
objRef, String Name)
{
    try {
    // get the root naming context
    org.omg.CORBA.Object obj = orb.resolve_initial_references("NameService");
```

```
    NamingContext rootContext = NamingContextHelper.narrow(obj);
    // Binding to NamingService
    System.out.println("Binding to NamingService");
    NameComponent nc = new NameComponent(Name, "");
    NameComponent path[] =
    {
        nc
    };
    rootContext.rebind(path, objRef);
    } catch (Exception ex) {
        System.out.println("Error in publishObjects  " + ex);
    }
}
public static org.omg.CORBA.Object activatePOAs(org.omg.CORBA.ORB orb,
Servant servantObj)
{
    org.omg.CORBA.Object obj = null;
    try {
        POA rootPoa = (POA)orb.resolve_initial_references("RootPOA");
        rootPoa.the_POAManager().activate();
        // Create a POA
        POA childpoa = null;
        // create policy for the new POA.
        Policy[] policy = new Policy[7];
        policy[0] = rootPoa.create_id_assignment_policy(
                    IdAssignmentPolicyValue.SYSTEM_ID);
        policy[1] = rootPoa.create_thread_policy(
                    ThreadPolicyValue.ORB_CTRL_MODEL);
        policy[2] = rootPoa.create_lifespan_policy(
                    LifespanPolicyValue.PERSISTENT);
```

```
            policy[3] = rootPoa.create_id_uniqueness_policy(
                        IdUniquenessPolicyValue.UNIQUE_ID);
            policy[4] = rootPoa.create_servant_retention_policy(
                        ServantRetentionPolicyValue.RETAIN);
           policy[5] = rootPoa.create_request_processing_policy(
                    RequestProcessingPolicyValue.USE_ACTIVE_OBJECT_MAP_ONLY);
           policy[6] = rootPoa.create_implicit_activation_policy(
                        ImplicitActivationPolicyValue.NO_IMPLICIT_ACTIVATION);
         // create the child poa and activate it
         childpoa = rootPoa.create_POA( "policy_2", null, policy );
         childpoa.the_POAManager().activate();
         childpoa.activate_object((Servant)servantObj);
         obj = childpoa.servant_to_reference((Servant)servantObj);
       } catch (org.omg.CORBA.ORBPackage.InvalidName ex) {
       } catch (org.omg.PortableServer.POAManagerPackage.AdapterInactive ex) {
       } catch (org.omg.PortableServer.POAPackage.AdapterAlreadyExists ex) {
       } catch (org.omg.PortableServer.POAPackage.InvalidPolicy ex) {
       } catch (org.omg.PortableServer.POAPackage.WrongPolicy ex) {
       } catch (org.omg.PortableServer.POAPackage.ServantAlreadyActive ex) {
       } catch (org.omg.PortableServer.POAPackage.ServantNotActive ex) {
           System.out.println("Error in activate POAs " + ex);
       }
       return obj;
   }
   public static org.omg.CORBA.ORB initializeORB(String orbId)
   {
       org.omg.CORBA.ORB orb = null;
       try{
           Properties prop = new Properties();
           prop.setProperty("org.omg.CORBA.ORBClass",
```

```
                    "com.sun.corba.se.internal.POA.POAORB");
        prop.setProperty( "com.sun.CORBA.ORBid", orbId);
        String[] initargs = {""};
        orb = ORB.init(initargs, prop );
        } catch (Exception ex) {
            System.out.println("caught Exception " + ex);
        }
        return orb;
}
public static void shutdown(org.omg.CORBA.ORB orb)
{
System.out.println("Server's shutdown method called");
}

public static void install(org.omg.CORBA.ORB orb)
{
    // could perform server specific installation, e.g.,
    // creating files, attaching to database, etc.
    System.out.println("Server's install method called");
}
public static void uninstall(org.omg.CORBA.ORB orb)
{
    System.out.println("Server's uninstall method called");
}
}
```

The server program here shows creation of multiple ORBs in a server process, and creating objects in those ORBs. The clients can invoke on an object in a particular ORB via the object reference published in the Namespace. The install(), uninstall(), and shutdown methods should be public static void(), and are invoked during server registration, unregistration, and shutdown.

2.4.3 Example Client Program

```
package examples;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.*;
public class policy2Client
{
    private static final String msgPassed = “policy_2: **PASSED**”;
    private static final String msgFailed = “policy_2: **FAILED**”;
    public static void main( String args[] )
    {
        try
        {
            Properties props = new Properties();
            props.put(“org.omg.corba.ORBClass”,
                    System.getProperty(“org.omg.CORBA.ORBClass”));
             props.setProperty( “com.sun.CORBA.ORBid”, “sunorb1”);
             System.out.println(“com.sun.CORBA.ORBid “ +
                     props.getProperty(“com.sun.CORBA.ORBid”));
             ORB orb1 = ORB.init( args, props );
             props = new Properties();
             props.put(“org.omg.corba.ORBClass”,
                    System.getProperty(“org.omg.CORBA.ORBClass”));
             props.setProperty( “com.sun.CORBA.ORBid”, “sunorb2”);
             ORB orb2 = ORB.init( args, props );
             lookupAndInvoke(orb1, “Object1”);
             lookupAndInvoke(orb2, “Object2”);
        } catch( Exception exp ) {
             exp.printStackTrace();
```

```
            System.out.println( msgFailed + "\n" );
        }
    }

    public static void lookupAndInvoke(org.omg.CORBA.ORB orb, String ObjName)
    throws Exception
    {
        try {
            System.out.println( "Looking for naming Service" );
            org.omg.CORBA.Object objRef =
                    orb.resolve_initial_references( "NameService" );
            NamingContext ncRef = NamingContextHelper.narrow( objRef );
            System.out.println( "Getting Object Reference" );
            NameComponent nc = new NameComponent( ObjName, "" );
            NameComponent path[] =
            {
                nc
            };
            policy_2 Ref = policy_2Helper.narrow( ncRef.resolve( path ) );
            int l = Ref.increment();
            System.out.println( "Incremented value:" + l );
            System.out.println( msgPassed + "\n" );
        } catch( Exception exp ) {
            throw exp;
        }
    }
    }
```

In this example, the client lookups the two objects created within different ORBs in a server, and invokes on them. As expected, since the objects are different, and have different bases for the increment method, the result is different for each invocation.

2.4.4 Makefile for the Example

```
 JAVA_HOME=/usr/local/java/jdk1.2.2/solaris

RIP_HOME=/net/anybodys/export3/anita/rip-int-apr5/build/solaris/

## NOT TO BE CHANGED

JAVA=$(JAVA_HOME)/bin/java

JAVAC=$(JAVA_HOME)/bin/javac

JDB=$(JAVA_HOME)/bin/jdb

CLASSPATH=$(RIP_HOME)/classes:.

JAVACFLAGS=-d . -classpath $(CLASSPATH)

.SUFFIXES: .class .java .java~

.java.class:

        $(JAVAC) $(JAVACFLAGS) $<

EXE_FLAGS = \

        -ORBInitialHost $(ORB_INITIAL_HOST) \

        -ORBInitialPort $(ORB_INITIAL_PORT)

VPATH=examples

STUBDIR=examples

client = policy2Client.class

server = policy2Server.class

LD_LIBRARY_PATH=$(RIP_HOME)/lib/sparc

IOSER=$(LD_LIBRARY_PATH)/libioser12.so

IDLJ=${JAVA} -classpath ${CLASSPATH} com.sun.tools.corba.se.idl.toJavaPortable.Compile

IDLJFLAGS=-fall -td . -verbose -i${JAVA_HOME}/lib -pkgPrefix CosTransactions org.omg

POAFLAGS=-poa

ORB_INITIAL_PORT=1050

ORB_INITIAL_HOST=anmol.eng.sun.com

SLEEP=/usr/bin/sleep

RM=/bin/rm -rf
```

```
ORB_CLASS=com.sun.corba.se.internal.POA.POAORB
ORBSINGLETON_CLASS=com.sun.corba.se.internal.corba.ORBSingleton
ORB_PROPS=-Dorg.omg.CORBA.ORBInitialHost=${ORB_INITIAL_HOST} \
        -Dorg.omg.CORBA.ORBInitialPort=${ORB_INITIAL_PORT} \
        -Dcom.sun.CORBA.ORBId=”sunorb”
JAVAFLAGS=$(ORB_PROPS) -classpath $(CLASSPATH)
ACTIVATION_DIR=.
ACTIVATION_PORT=1049
ORBD_CLASS=com.sun.corba.se.internal.Activation.ORBD
ORBD_PROPS=-Dcom.sun.CORBA.Activation.DbDir=$(ACTIVATION_DIR) \
            -Dcom.sun.CORBA.Activaton.Port=$(ACTIVATION_PORT)
ORBD=$(JAVA) $(ORBD_PROPS) $(JAVAFLAGS) $(ORBD_CLASS)
SERVERTOOL_CLASS=com.sun.corba.se.internal.Activation.ServerTool
SERVERTOOL=$(JAVA) $(JAVAFLAGS) $(SERVERTOOL_CLASS)
all : clean build run
stubs: $(STUBDIR)/policy_2.java
build: stubs $(client) $(server)
run: register start runclient
#
# Targets to compile the tests.
#
$(STUBDIR)/policy_2.java: policy2.idl
    $(IDLJ) $(IDLJFLAGS) $(POAFLAGS) policy2.idl
#
# Target to register the server.
#  (note that it will be put in the background)
#
register:
    $(SERVERTOOL) -ORBInitialPort $(ORB_INITIAL_PORT) -cmd \
    register -server examples.policy2Server \
```

```
    -applicationName s1 \
    -vmargs \
    -Dorg.omg.CORBA.ORBClass=com.sun.corba.se.internal.POA.POAORB \
    -Dorg.omg.CORBA.ORBInitialPort=1050 >out
    grep serverid out|cut -f2 -d= >out
    $(SLEEP) 2
#
# Target to start the server.
#  (note that it will be put in the background)
#
start:
    $(SERVERTOOL) -ORBInitialPort $(ORB_INITIAL_PORT) -cmd \
    startup -serverid `cut -c 2,2-4 out` >out
    rm -f out
    $(SLEEP) 3


#
# Targets to run the client.
#
runclient:
    $(JAVA) -Dorg.omg.CORBA.ORBClass=${ORB_CLASS} \
     -Dorg.omg.CORBA.ORBSingletonClass=${ORBSINGLETON_CLASS} \
     $(JAVAFLAGS) examples.policy2Client $(EXE_FLAGS)
runserver:
    $(JDB) -Dcom.sun.CORBA.POA.ORBServerId=4000 \
    -Dcom.sun.CORBA.POA.ORBPersistentServerPort=1050 \
    -Dcom.sun.CORBA.ORBId=sunorb \
          -Dcom.sun.CORBA.activation.DbDir=$(ACTIVATION_DIR)  $(JAVAFLAGS)
examples.policy2Server $(EXE_FLAGS)
runorbd:  $(ORBD)
```

clean:

$(RM) examples

$(RM) NC0 counter poaids.db servers.db logs orb.db

servertool:

$(SERVERTOOL) -help

2.4.4.1 Building the Programs

make build

This command generates the stubs and skeletons, and builds the programs. The class files are placed under examples, in the current directory.

2.4.4.2 Executing the Programs

make runorbd

This command starts up the orbd

make servertool

This command starts up the servertool. This servertool can then be used to startup servers, etc. The register target in the makefile can be used as a reference for registering policy2Server with the ORBD. Once the server is registered, the policy2Client can be executed using the command:

make runclient

The orb.db directory in the current directory contains the persistent data and the server logs. The orb.db/logs directory contains the <serverid>.out and <serverid>.err file for each server that is registered through servertool. The application programmer may refer to this directory for finding out any messages from their servers.

Portable Server Activation 3

3.1 Introduction

It has become obvious that our current server activation framework (SAF) has some serious problems that need to be fixed. The major problems are:

1. It is not currently possible to write the SAF using only public CORBA APIs. The major problem here is a lack of a means to do what we call object reference template exchange. The object reference template RFP is aimed at changing this.
2. Our old SAF does not correctly handle policies on objects, which are typically expressed as tagged components in IORs. The IOR interceptor both makes this problem worse, and provides part of the tools needed to fix it. This is why we are introducing the object reference template.
3. The old SAF has some problems with concurrency control, including:
   a. It does not detect when a server instance fails!
   b. It does not handle race conditions with POA startup. Consider the case where a persistent object reference causes a server to be restarted. We need to wait for the following events:
      i. The server is running. This is in the code now.
      ii. The ORB has registered its endpoints. This is missing, and is a bug today.
      iii. The POA is/will be available. This is interesting because of adapter activators. This is discussed below in Section 3.4.5, “POA Startup Problems,” on page xxxi.

The following sections will discuss our goals for a new SAF and sketch the solution to the above problems.

3.2 Goals

There are a number of aspects to consider in designing an activation framework including performance, reliability, simplicity, and use of resources. Our motivation in designing a Server Activation Framework (SAF) include a desire to have a reasonably simple and clean implmentation and to build an entirely portable framework. Accordingly, here is a list of our goals:

1. The SAF should be built only on public APIs (OMG and Java). The OMG APIs include the POA, Portable Interceptors, and the evolving Object Reference Template specification. On the Java side, we can use any public APIs in J2SE 1.3.

2. The SAF should rely on a minimal amount of persistent state. In fact, the only persistent state should be what is needed to start a sever. All other state comes from the servers themselves through a series of registration processes. Note that this means that we will store ORB ID strings and POA name sequences directly in object keys, and remove the translation to integers that is present today.
3. Simplicity is more important than reliability. Our design will use a single ORBD per host, where the ORBD combines the implementation repository and location/activation functions.
4. This must be built fairly quickly, so we will use the existing ORBD implementation and portableactivation prototype as starting points.
5. The new SAF must correctly handle all POA activation scenarios.
6. The new SAF must implement at least rudimentary monitoring of servers so that we can fix some known problems with server recovery.
7. Some consideration should be given to instrumentation for monitoring and debugging.
8. We will assume that the ORBD never crashes. If it does, all information about running servers will be lost. Obviously this could be fixed, but the implementation would be more complex, requiring committment of significant state changes to stable storage, and possibly a mechanism for restarting ORBD after it crashes. These are not features we want to implement right now.

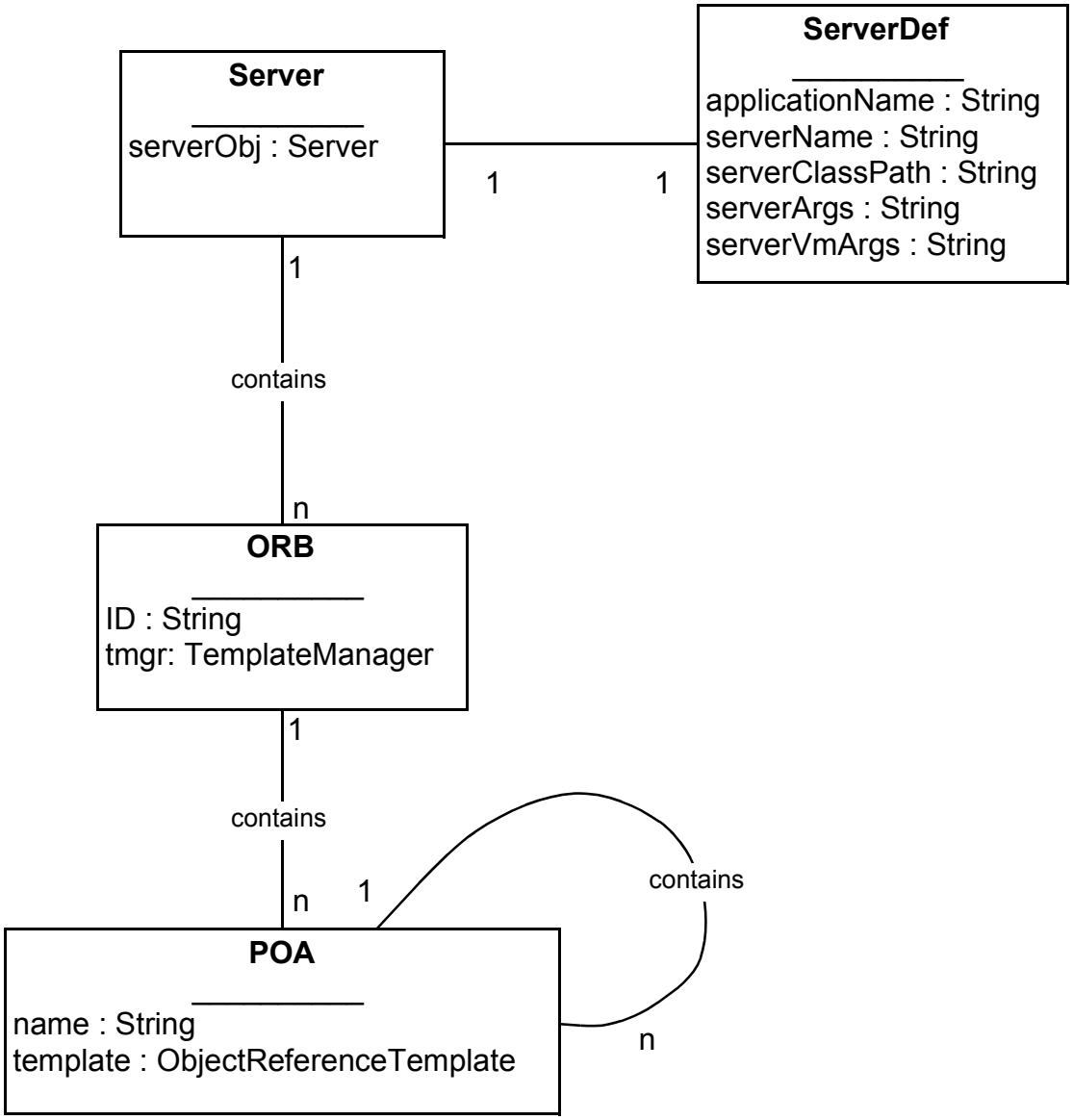
3.3 IDL interfaces for SAF

3.4 Scenarios for SAF

There are a number of problems that a SAF must solve to correctly support the current CORBA specifications. The following subsections give the scenarios and include bits of the design.

3.4.1 Data model for server state

The SAF must track the states of Servers, ORBs, and POAs. The data required is roughly as follows:



There are a number of locks and condition variables that are not shown here. All of the required data must be collected as needed:

a. The ServerDef is created and managed using ServerTool.

b. The ServerMain wrapper class must implement the Server object and register it with the activator. The ServerMain wrapper can also report when the launched main class completes.

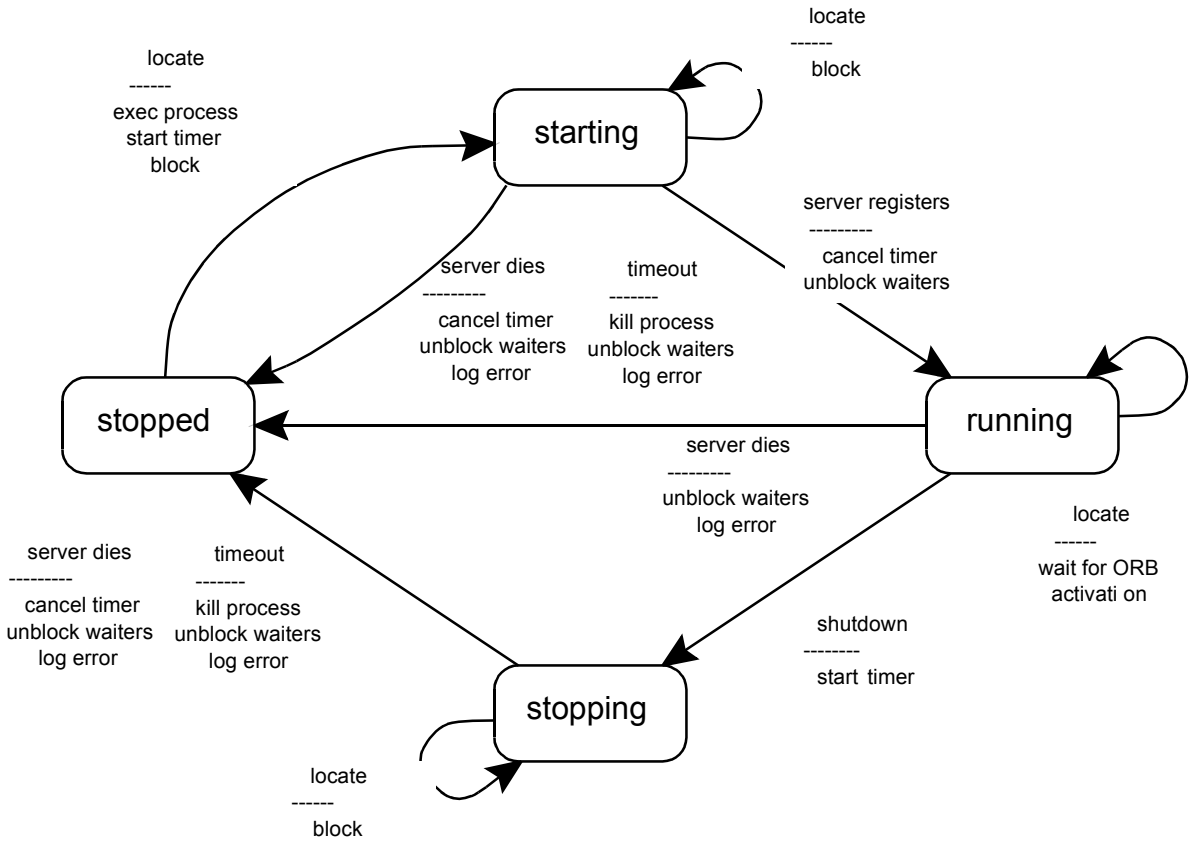
c. Each ORB must have an ORB intializer that registers all required server interceptors and also registers the ORB with ORBD. The initializer can also register some kind of interceptor (type probably doesn't matter) whose destroy

method informs ORBD when the ORB is terminating. We can force this interceptor to be registered in every ORB by putting the initializer property name into the system environment. This can be forced by passing the correct command line argument to ServerMain.

d. The server must have a template interceptor that registers the POA's template with the ORBD. We will probably also add a POA destruction interceptor that can inform the ORBD that a POA is no longer in use.

3.4.2 Monitoring Server State

The SAF must monitor the state of each server that it launches. In particular, it must know about key state transitions in the lifecycle of a server. We will implement this according to the following state machine:



3.4.3 Server Activation on Demand

The SAF must be able to launch a server on demand from a client invocation.

3.4.4 Server Lifecycle

The SAF must support installation, startup, shutdown, and removal of servers. This is already present in the old SAF, and I anticipate no change here. Note that once a server is uninstalled, the same server can never be re-installed. This all object reference for the old instance become invalid.

3.4.5 POA Startup Problems

The SAF must correctly handle POA startup.

The scenario must work as follows:

1. Client invokes ORBD, server is not running.
2. ORBD starts server; client invoke is blocked waiting for server.
3. server starts up and registers with ORBD; client unblocked, moved to wait for ORB registration.
4. ORB registers with ORBD; client unblocked, but now waits for POA to register. At this point, ORBD sends template_required to ORB's TemplateManager object, giving it the required POA name sequence.
5. TemplateManager gets template_required, and calls find_POA on the POA name sequence starting at the root POA. Three cases can occur here:
   a. The sequence has adapter activators all the way, so the find_POA call causes the POA to be created, and thus registered with the ORBD.
   b. Insufficient adapter activators are present, but server initialization eventually causes the POA to be created, so the registration occurs.
   c. Insufficient adapter activators are present, and server initialization incorrectly fails to initialize the required POAs. In this case, the client invocation must eventually timeout and fail.
   d. POA is created in initialiation sequence, registers with ORBD. Now ORBD can correctly construct the forwarded IOR, since it has a template.

We could certainly put timeouts on the waits if desired.

3.4.6 Object Reference Policies

The policies on an object reference that were determined by create_POA must also be present on the object reference created by the ORBD. This is the whole point of the template.

3.4.7 Access to Template Policies

If necessary, an object reference create in the ORBD may inspect a registered server template and create corresponding policies.

3.4.8 Opening the Framework

A server started with no special arguments will NOT participate in the activation framework described here.

3.4.9 Mapping Reqiest to Templates

An incoming request must be mapped to the corresponding server object template.

We will take a simple approach to solve this by encoding all needed information into the POA name of the ORBD POA for the ORBD object corresponding to the server object. The name of the ORBD POA will be

<ServerID> / <ORBID> / <POAName>

and the object ID will be the same in both the ORBD and the Server objects. This then allows the ORBD to determine what Server to start, what ORB to wait for, and what POA to both invoke template_required on and to wait for.

3.5 Structure of our Object Reference Template

3.6 Design choices for endpoint association

Existing ORBs differ greatly on which entities are associated with transport endpoints. Known choices include:

1. Endpoints belong to server.

   This is essentially the way Sun ORBs have worked until recently. This does not really work properly, since each ORB instance is an independent entity. The advantage of course is that fewer resources are consumed in servers.

2. Endpoints belong to ORB.

   This is the Sun model. This is a resonable choice for resource consumption. However, correctly handling POAManager state transitions can be more complex than in some other choices.

   The problem is that a POAManager in the holding state must queue requests somewhere. Minimizing the consumption of resources while queuing requests is a good goal. Our current implementation while hold onto a lot of state at this point: basically an entire server thread will be blocked while a POAManager is in the holding state. Long fragmented requests are a problem too, since the fragments must be accepted. We probably should modify our ORB to return transient errors if too much data is queued while a POAManager is holding. Note that we cannot properly force flow control into the client with the ORB model.

3. Endpoints belong to POAManager.

This is the choice taken in the OOC ORBs. Here the POAManager holding state can be efficiently implemented simply by allowing the socket to queue data until the OS decides to stop accepting more data, effectively flow controlling the client. Client flow control is in fact not possible with the ORB model.

Implementing this model also requires interceptor support for POAManager state changes (I think).

4. Endpoints belong to POA. This is apparently what ORBIX 2000 does (according to Michi Henning).

Extending Server Activation 4

Not yet written.

The Implementation Repository 5

Not yet written.