HPSA Extension Pack

SOSA Developer Reference

Release v.5.1

Legal Notices

Warranty.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

A copy of the specific warranty terms applicable to your Hewlett-Packard product can be obtained from your local Sales and Service Office.

Restricted Rights Legend.

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Hewlett-Packard Company United States of America

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

Copyright Notices.

©Copyright 2001-2009 Hewlett-Packard Development Company, L.P., all rights reserved.

No part of this document may be copied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this material is subject to change without notice.

Trademark Notices.

Java™ is a U.S. trademark of Sun Microsystems, Inc.

Linux is a U.S. registered trademark of Linus Torvalds

Microsoft® is a U.S. registered trademark of Microsoft Corporation.

Oracle® is a registered U.S. trademark of Oracle Corporation, Redwood City, California.

UNIX® is a registered trademark of the Open Group.

Windows® and MS Windows® are U.S. registered trademarks of Microsoft Corporation.

All other product names are the property of their respective trademark or service mark holders and are hereby acknowledged.

Document id:

Table of Contents

Support

Support for the HP Service Activator Extended Pack product is available on the following mailing list:

hpsa-support@hp.com

In This Guide

This guide describes the Sosa product, including its architecture and use.

Audience

The audience for this guide is the Solutions Integrator (SI). The SI has a combination of some or all of the following capabilities:

Understands and has a solid working knowledge of:

- UNIX® commands
- Windows® system administration

Understands networking concepts and language

Is able to program in Java™ and XML

Understands security issues

Understands the customer's problem domain

References

HPSA Extension Pack - SOSA3 - Management.doc

Conventions

The following typographical conventions are used in this guide.

Font	What the Font Represents	Example
Italic	Book or manual titles, and man page names	Refer to the *HP Service Activator — Workflows and the Workflow Manager* and the *Javadocs* man page for more information.
	Provides emphasis	You *must* follow these steps.
	Specifies a variable that you must supply when entering a command	Run the command: `InventoryBuilder` *<sourceFiles>*
	Parameters to a method	The *assigned_criteria* parameter returns an ACSE response.
Bold	New terms	The **distinguishing attribute** of this class...
`Computer`	Text and items on the computer screen	The system replies: `Press Enter`
	Command names	Use the `InventoryBuilder` command ...
	Method names	The `get_all_replies()` method does the following...
	File and directory names	Edit the file `$ACTIVATOR_ETC/config/mwfm.xml`
	Process names	Check to see if `mwfm` is running.
	Window/dialog box names	In the `Test and Track` dialog...
	XML tag references	Use the `<DBTable>` tag to...
`Computer Bold`	Text that you must type	At the prompt, type: **`ls -l`**
Keycap	Keyboard keys	Press **Return**.
[Button]	Buttons on the user interface	Click `[Delete]`. Click the `[Apply]` button.
Menu Items	A menu name followed by a colon (:) means that you select the menu, then the item. When the item is followed by an arrow (->), a cascading menu follows	Select Locate:Objects->by Comment.

Install Location Descriptors

The following names are used throughout this guide to define install locations.

Descriptor	What the Descriptor Represents
$ACTIVATOR_OPT	The install base location of Service Activator. The UNIX location is `/opt/OV/ServiceActivator` The Windows location is `<drive>:\HP\OpenView\ServiceActivator\`
$ACTIVATOR_ETC	The install location of specific Service Activator configuration files. The UNIX location is `/etc/opt/OV/ServiceActivator` The Windows location is `<drive>:\HP\OpenView\ServiceActivator\etc\`
$ACTIVATOR_VAR	The install location of specific Service Activator logging files. The UNIX location is `/var/opt/OV/ServiceActivator` The Windows location is `<drive>:\HP\OpenView\ServiceActivator\var\`
$ACTIVATOR_BIN	The install location of specific Service Activator binary files. The UNIX location is `/opt/OV/ServiceActivator/bin` The Windows location is `<drive>:\HP\OpenView\ServiceActivator\bin\`
$ACTIVATOR_THIRD_PARTY	The location for new Java components such as workflow nodes and modules. Third-party libraries can also be placed in this directory. The UNIX location is `/opt/OV/ServiceActivator/3rd-party` The Windows location is `<drive>:\HP\OpenView\ServiceActivator\3rd-party\` Customized inventory files are stored in the following locations: UNIX: `$ACTIVATOR_THIRD_PARTY/inventory` Windows: `$ACTIVATOR_THIRD_PARTY\inventory`
$JBOSS_HOME	HOME The install location for JBoss. The UNIX location is `/opt/HP/jboss` The Windows location is `<drive>:\HP\jboss`
$JBOSS_DEPLOY	The install location of the Service Activator J2EE components. The UNIX location is `/opt/HP/jboss/server/default/deploy`

	The Windows location is `<drive>:\HP\jboss\server\default\deploy`
`$ACTIVATOR_DB_USER`	The database user name you define. Suggestion: `ovactivator`
`$ACTIVATOR_SSH_USER`	The Secure Shell user name you define. Suggestion: `ovactusr`
`$SOSA_HOME`	The install base location of SOSA. The default UNIX location is `/opt/OV/Sosa` The default Windows location is <drive>:\HP\OpenView\Sosa\
`$SOSA_BIN`	The install location of specific SOSA binary files. The default UNIX location is `/opt/OV/Sosa/bin` The default Windows location is `<drive>:\HP\OpenView\Sosa\bin\`
`$SOSA_CONFIG`	The install location of specific SOSA configuration files. The default UNIX location is `/opt/OV/Sosa/config` The default Windows location is <drive>:\HP\OpenView\Sosa\config\
`$ECP_HOME`	The install base location of Equipment Connections Pool. The default UNIX location is `/opt/OV/ECP` The default Windows location is <drive>:\HP\OpenView\ECP\
`$ECP_BIN`	The install location of specific Equipment Connections Pool binary files. The default UNIX location is `/opt/OV/ECP/bin` The default Windows location is <drive>:\HP\OpenView\ECP\bin\
`$ECP_ETC`	The install location of specific Equipment Connections Pool configuration files. The default UNIX location is `/opt/OV/ECP/conf` The default Windows location is <drive>:\HP\OpenView\ECP\conf\

1 Introduction

1.1 Purpose

This document is meant as a developers reference guide for the SOSA latest version. It contains all the information about this application, its features and how to use them.

1.2 Acronyms

SOSA: Service Order Smart Adapter

MWFM: Micro Work Flow Manager

HPSA: HP Service Activator

HPSA EP: HPSA Extension Pack

SO: Service Order

SA: Service Action

SAE: Service Action Executor

PA: Protocol Adapter

2 Introduction to SOSA

This chapter presents an introduction to an overview of SOSA.

2.1.1 Functinality

Service Order Smart Adapter (SOSA) is a flexible adapter to manage the influx of Service Orders, which are aimed at the transactional activation engine called Service Activator. In this way, SOSA provides additional features for the treatment of these requests compared to a traditional system.

The main features added are:

The possibility to receive incoming requests to the system in multiple formats, thanks to the capacity of SOSA to implement new Protocol Adapters that can receive requests in any known format.

The possibility to define a flexible service catalog, which allows decomposing each service into a list of provision flows to be launched for each Service Order.

Guaranteed transactional of the Service Order.

Possibility to define service queues to allow better management when there are problems with the system's performance.

Modular administration of Service Order persistence and historic records.

2.1.2 Architecture

Figure 1 shows the major components of the architecture of Sosa. This section only scrapes the surface. For more information about the components, go to the user reference manual.

Protocol Adapters are the components which listen to and handle the queries sent by the external clients and they build the Service Order trees that will be inserted into SOSA to be processed. SOSA, on its installation, provides two protocol adapters for working with queries via RMI and Web Services.

When a Service Order is inserted into SOSA, the Service Order Processor and the Service Action Processor are the components that handle that service through its execution, changing its status on each step and responding to the Protocol Adapter in the end.

Queues are responsible for storing Service Actions until a consumer takes them out to be executed by a Service Activation Executor. Each queue can have several consumers. Two queues are provided by default, a basic (FIFO) queue and a priority queue.

The Service Action Executor executes Service Actions loaded by a queue consumer. Executors can work in two ways: synchronous or asynchronous modes. Since most of the Service Actions represent an HPSA workflow execution, SOSA provides a concrete Service Action Executor to work with these workflows ‘the MWFM Service Action Executor’. The communication between the executor and the HPSA is performed via RMI.

The Service Catalog stores the needed information to build the whole service tree associated with a Service Order. The Service Catalog implementation provided by SOSA is based on JDBC and stores all the elements in an Oracle data base.

The SOSA Persistence allows keeping a non-volatile register of the services which are being executed, in order to prevent the loss of received queries due to system shutdown. SOSA is provided with a persistence based on Hibernate, an object/relational tool.

Also, it is possible to configure a History manager in SOSA. This is the manager that is used to listen to the end of Service Orders, which are processed and responded to the client, and then it moves them to the History register. Like Persistence, the History is based on Hibernate.

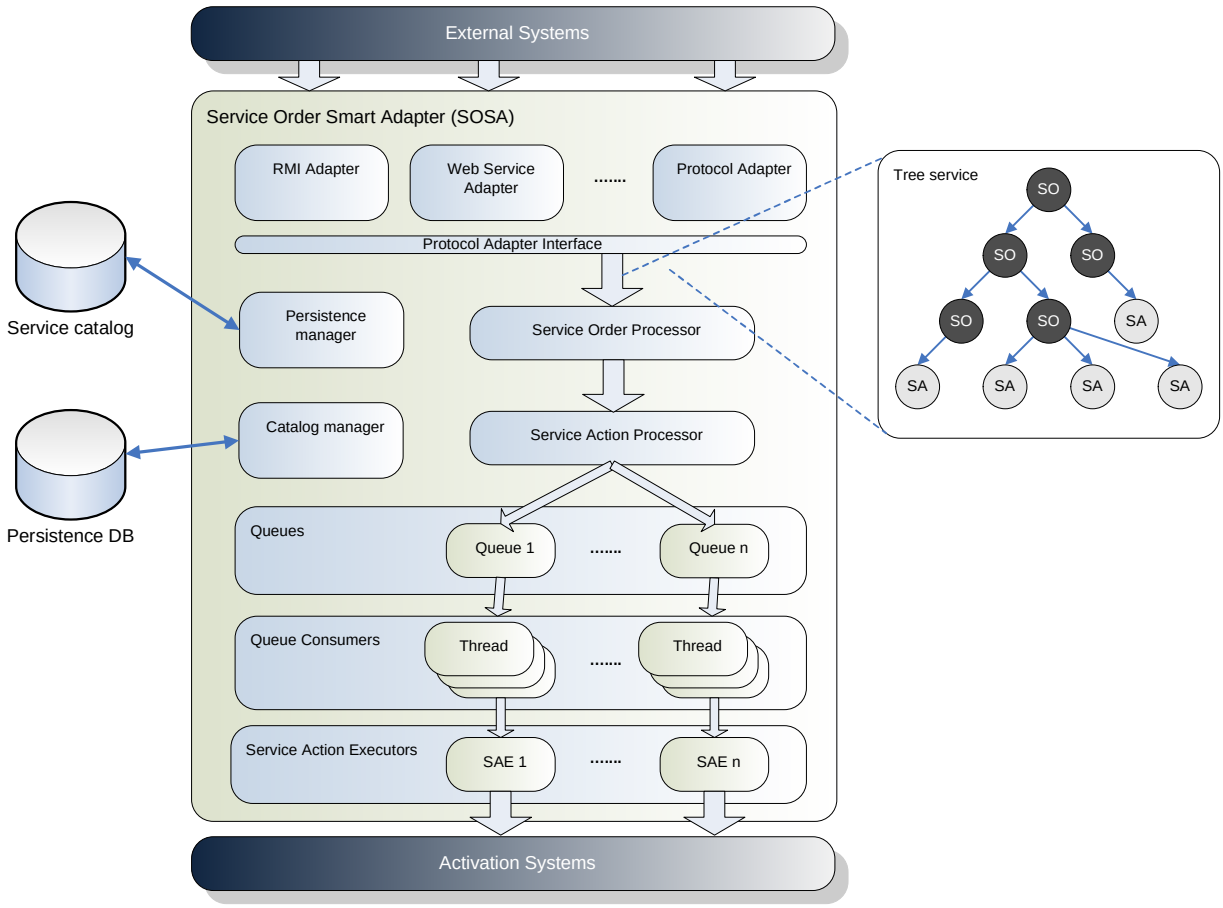

Fig. 1: SOSA architecture

2.1.3 Modules

The SOSA installation configures two modules: the Main SOSA Module and the Time Window Module

2.1.3.1 Main Module

When SOSA starts the Main Module initialize all SOSA components, managers, protocol adapters, service catalog, persistence, etc. Also the module is responsible for stopping all the components when SOSA is shutting down.

The module is configured in the SOSA module configuration file, at $SOSA_CONFIG/`sosa.xml` and needs at least the next parameters:

`sosa.conf.dtd.file`: indicates the path of SOSA DTD configuration file.

`sosa.conf.file`: indicates the path of SOSA configuration file.

```
<Modules>
  <Module name="sosaModule" className="com.hp.sosa.modules.sosamodule.SosaModule">
    <Parameter name="sosa.conf.dtd.file" value="conf/sosa_conf.dtd" />
    <Parameter name="sosa.conf.file" value="conf/sosa_conf.xml" />
  </Module>
</Modules>
```

2.1.3.2 Time Window Module

The Time Window Module offers a way to schedule tasks or actions to be applied to several SOSA components. Later in this document, the Time Window Module functionality will be explained in detail.

As the Main Module, the Time Window Module is configured in the SOSA module configuration file and basically contains the data base parameters where the scheduling task will be stored:

```
<Modules>
  <Module name="timeWindowModule"
className="com.hp.sosa.modules.timewindowmodule.TimeWindowModule">
    <Parameter name="db.pool.name" value="db_time_window_module" />
    <Parameter name="db.user" value="userpp" />
    <Parameter name="db.password" value="userpp" />
    <Parameter name="db.jdbc.driver" value="oracle.jdbc.driver.OracleDriver" />
    <Parameter name="db.driver.name" value="jdbc:oracle:thin" />
    <Parameter name="db.host" value="127.0.0.1" />
    <Parameter name="db.port" value="1521" />
    <Parameter name="db.instance" value="HPSA" />
    <Parameter name="db.initialsize" value="2" />
    <Parameter name="db.maxactive" value="4" />
    <Parameter name="db.maxidle" value="4" />
    <Parameter name="db.minidle" value="0" />
    <Parameter name="db.maxwait" value="2000" />
  </Module>
</Modules>
```

2.1.4 Starting and Stopping SOSA

After installing or updating SOSA through the HPSA Extension Pack Installer, the SOSA module and all its provided components (protocol adapters, service catalog, persistence…) are ready to work.

These are the commands to manage the SOSA instance:

To start SOSA: $SOSA_BIN/`sosa.sh start`

To stop SOSA: $SOSA_BIN/`sosa.sh stop`

To restart SOSA: $SOSA_BIN/`sosa.sh restart`

To test SOSA: $SOSA_BIN/`sosa.sh test`

3 Service Action Executors

Service Action Executors (SAEs) are the elements in charge of executing the Service Actions on the selected service activation platform. All implemented SAEs must be able to work in both synchronous and asynchronous mode.

Synchronous mode: the executor sends the Service Action to be executed and waits until processed. Then it must return the response to the Service Action Processor.

Asynchronous mode: the executor sends the Service Action to be executed without waiting until processed. The service activation platform must notify SOSA the execution finalization through an RMI service. Of course, the SAE must tell the final system the URL of that RMI service.

Usually, the synchronous mode consumes more resources than asynchronous due to threads remains waiting for the execution completion. Therefore, in most cases asynchronous mode is the best choice.

The way of Service Actions are executed by the SAE depends on the queue where they are processed. Synchronous queues will process Service Actions in a synchronous mode and asynchronous queues will process in an asynchronous mode. The next chapter in this document explains queues in detail.

SOSA can initialize as many SAEs as desired. Internally, SOSA has a Service Action Execution Handler which manages the different executors running on SOSA. The handler initializes the executors using the configuration found on SOSA configuration file at $SOSA_CONFIG/`sosa_conf.xml`, inside the `<ServiceActionExecutors>`.

Each SAE is represented by the `<ServiceActionExecutor>` tag, which has the following attributes:

`name`: the name of the Service Action Executor.

`className`: the complete class name of this Service Action Executor implementation.

`max_parallelism`: the maximum number of concurrent executions that can be handled by the executor.

The configuration parameters of each Service Action Executor can be included inside the `<ServiceActionExecutor>` tag using the `<Parameter>` tag, which has a `name` and a `value` attributes. Next section shows an example configuration of a SAE.

SOSA provides an abstract class, `ServiceActionExecutors`, which allows developers to build new SAEs.

3.1.1 MWFM Service Action Executors

In many of SOSA scenarios, Service Actions represents HPSA workflows. The MWFM Service Action Executor is a SAE implementation provided by SOSA, which is able to manage the workflow execution on a HPSA MWFM instance.

The communication between the MWFM SAE and the MWFM instance is performed via RMI. When MWFM SAE starts a workflow use the RMI interface of the MWFM. If the execution is in asynchronous mode, MWFM will have to notify MWFM SAE at the end of execution. Therefore, asynchronous mode need that MWFM knows how to communicate with SAE. Later we will see how to do this.

To add a MWFM SAE to SOSA a new entry in the SOSA configuration file must be added. Below is shown an example configuration of a MWFM SAE.

```
<ServiceActionExecutors>
  <ServiceActionExecutor
className="com.hp.sosa.modules.sosamodule.executors.mwfm.MwfmServiceActionExecutor"
max_parallelism="0" name="MWFM_SA_EXECUTOR">
    <Parameter name="host" value="127.0.0.1"/>
    <Parameter name="port" value="2000"/>
    <Parameter name="user" value="admin"/>
    <Parameter name="password" value=""/>
    <Parameter name="async_interval" value="60"/>
    <Parameter name="async_queue" value="sosa_async_queue"/>
    <Parameter name="launch_retries" value="1"/>
    <Parameter name="copy_cp_to_output" value="false"/>
    <Parameter name="timeout" value="90000"/>
    <Parameter name="timeout_interval" value="30000"/>
  </ServiceActionExecutor>
<ServiceActionExecutors>
```

The MWFM SAE configuration parameters are the following:

Name	Description
host	It indicates the host where the MWFM is running.
port	It indicates the port where the RMI instance of MWFM is published.
user	It indicates the user name of the MWFM.
password	It indicates the password of the MWFM user.
launch_retries	It indicates the number of times the executor will retry if a Service Action cannot be sent for any reason.
async_queue	It indicates the name of the MWFM message queue on which the asynchronous Service Actions notifications will be written in case the RMI service provided by the asynchronous response module is unavailable.
async_interval	It indicates the interval for checking the `async_queue` looking for asynchronous Service Actions finalization.
max_time_sync_job	Max time that a workflow launched synchronously in MWFM can run before to interrupt.
max_time_sync_job_interval	The interval for checking whether a workflow launched asynchronously has spent the `max_time_sync_job`.
timeout	It indicates the maximum amount of processing time for a Service Action to consider it timed out.
timeout_interval	It indicates the minimum time between timeout checks.
copy_cp_to_output	It indicates whether to copy all workflow case-packet variables into the `Output_Params` map when the execution finishes.

add_extra_info	In case the ServiceAction's extra_info parameter has value, add the value to the input params. The extra info parameter is a string and the ServiceActionExecute transform to a HashMap. For example, the string variable1=value1|;|variable2=value2 is transform to hashmap {variable1=value1,variable2=value2}
extra_info_value_limiter	Default value "="
extra_info_attrib_limiter	Default value |;|

3.1.2 SOSA Asynchronous Response Module

When the MWFM SAE is executing Service Actions in asynchronous mode, the workflow needs a way to notify SOSA from HPSA MWFM when it is finished. SOSA provides this tool as a HPSA MWFM module called SOSA Asynchronous Response Module.

Fist of all, the module has to be configured in the MWFM configuration file $ACTIVATOR_ETC/`mwfm.xml` as follows:

```
<Module>
  <Name>sosa_async_responser</Name>
  <Class-Name>com.hp.spain.engine.module.sosa.SosaAsyncResponserImpl</Class-Name>
  <Param name="errors_async_persistence_file"
value="C:/hp/OpenView/ServiceActivator/var/tmp/errors_async_responser.dat"/>
  <Param name="write_in_queue" value="true"/>
  <Param name="sosa_async_queue" value="sosa_async_queue"/>
  </Module>
```

Where:

Name	Description
errors_async_persistence_file	It is a mandatory parameter that indicates the path and the name of the persistence file used if the communication with SOSA fails.
write_in_queue	It indicates whether it a message will be sent to a queue if the communication with SOSA fails.
sosa_async_queue	The name of the queue where messages will be sent.
rmi_port	The port where publish a rmi service. Default 2000

The SOSA Asynchronous Response Module operation is as follows:

First, the module tries to notify SOSA through RMI connection.

If RMI connection fails, the module tries to write a message in the MWFM queue in case of it is configured.

If writing messages fails or it is not configured, the module stores the response in persistent file to retry the response later.

3.1.3 Developing workflows compatible with Sosa

Once we have seen how SOSA invokes HPSA workflows using the MWFM SAE, and how the MWFM is able to response SOSA via the Asynchronous Response Module, it is time to see how the HPSA workflows have to be developed to be compatible with SOSA.

Two main points have to be taken into account in respect to the workflow compatibility with SOSA. On one hand the workflow has to be able to invoke the response module, on the other hand a set of case-packet variables has to be defined in order to receive and sent parameters from or to SOSA, respectively.

3.1.3.1 Invoking the SOSA Asynchronous Response Module from a workflow

SOSA provides two ways to invoke the response module, a MWFM node and a MWFM end-handler.

The recommended way is to configure the SosaEndHandler, because the EndHandler will execute even if the workflow fails.

```
<End-Handler>
      <Name>SosaEndHandler</Name>
      <Class-Name>com.hp.sosa.endhandler.SosaEndHandler</Class-Name>
</End-Handler>
```

The other and not recommended is using a MWFM node `EndAsyncJob. This node` can be used like another MWFM node in a workflow. When the workflow runs this node, it contacts with the module to response to SOSA. The node can receive an optional parameter `queue` to notify the end of an execution using the MWFM message queue when the RMI service fails.

```
</Process-Node>
  <Process-Node>
  <Name>EndAsyncJob</Name>
  <Action>
    <Class-Name>com.hp.sosa.nodes.EndAsyncJob</Class-Name>
  </Action>
</Process-Node>
```

3.1.3.2 Related SOSA case-packet variables in workflows

SOSA use the following case-packets variables to interact and work with workflows:

Name	Description
S_ACTION	DO or UNDO. UNDO means the Service Action failed and it is executing the rollback workflow. DO means the Service Action is executing its usual operation.
S_SERVICE_NAME	The service name of the SA as defined in the service catalog.
S_SERVICE_ACTION	The service action of the SA as defined in the service catalog.
S_INPUT	The map containing the input parameters. SOSA copies in this map the input parameters of the Service Order, so it can be used in the workflow.
S_ROLLBACK	The map containing the rollback information. The workflow can put needed information for rollback in this map, so it can be used in the rollback workflow if necessary.
S_CONTEXT	The map containing the Service Order context. All the workflows executed by the Service Actions belonging to the same Service Order can access this map to share parameters.
S_OUTPUT	The map containing the output parameters. The workflows can put the output parameters in this map.
S_SOSA_CODE	It is the way to notify Sosa if the workflow finished correctly (value 0) or not (value different from 0).
S_SOSA_DESCRIPTION	The description of the workflow execution.
S_CODE	User defined code.
S_DESCRIPTION	User defined description.
S_FORCE_RETRY	Whether force a retry of the workflow in case on error.

3.1.4 Developing ServiceActionExecutor

Sometimes is required to develop a new ServiceActionExecutor because SOSA has to start actions in a different external system than MWFM. To develop a new ServiceActionExecutor is useful to follow next steps.

Let's suppose that the new class ServiceActionExecutor is named NewExampleServiceActionExecutor. This new example class has to extend the com.hp.sosa.modules.sosamodule.executors.ServiceActionExecutor class.

```
public class NewExampleServiceActionExecutor extends ServiceActionExecutor
```

After that, it will be some methods that are required to implement:

public boolean check() : This method return true in case the ServiceActionExecutor has connectivity and is able to start orders correctly into the external system. In case, the connectivity is down, the external system is down, … this method has to return false.

public void finish() : When SOSA is stopping this method will be called and then the ServiceActionExecutor can disconnect or un-configure the required variables or environment.

public boolean haveToWaitAsynchronousResponse(String ssid) : when SOSA starts and detects a ServiceAction was executing in asynchronous mode on this instance of ServiceActionExecutor, SOSA calls this method to know if it can recover or not the ServiceAction in processing status. Return true if this particular ServiceAction is possible to keep on processing status because it's sure SOSA will receive the answer of this order. Return false in the rest of the cases.

public boolean haveToWaitSynchronousResponse(String ssid) : when SOSA starts and detects a ServiceAction was executing in synchronous mode on this instance of ServiceActionExecutor, SOSA calls this method to know if it can recover or not the ServiceAction in processing status. Return true if this particular ServiceAction is possible to keep on processing status because it's sure SOSA will receive the answer of this order. Return false in the rest of the cases.

public void init(Map initParmeters) : the initParameters map contains all the parameters configured into the sosa_conf.xml file for this ServiceActionExecutor. This method will be called by SOSA when start and before to send any ServiceAction to the executor to be able to configure it.

public void refreshConfiguration(Map initparms) : the initParameters map contains all the parameters configured into the sosa_conf.xml file for this ServiceActionExecutor. This method will be called by SOSA when the configuration needs to be refreshed. In case is not possible to refresh the configuration leave empty this method.

public void killService(String ssid) : in case the ServiceAction has configured a maximum time to execute, this method will be called when the ServiceAction has timeout and the executor should kill the action into the external system if it's possible.

public ServiceActionResponse process(String ssid) : when the ServiceAction is going to execute as synchronous this method will be called. There are two main task to do, execute the order into the external system and return the result via ServiceActionResponse.

public ServiceActionResponse[] process(String[] ssids) : some queues are able to execute ServiceAction in groups. This method will be called in case the queue support and is configured to

execute groups of ServiceAction as synchronous. If the external system doesn't not support groups execution then it's recommended just to make next loop.

```
if (ssids != null){
  ServiceActionResponse[] sars = new ServiceActionResponse[ssids.length];
  for (int i=0; i<ssids.length;i++){
      sars[i] = process(ssids[i]);
  }
  return sars;
}
```

public void processAsync(String ssid, String rmiUrl) : this method will be called when the ServiceAction will be executed as sysnchornous. This means the external system has to connect to SOSA to return the result. Then this method only will start the order into the external system. By default SOSA provides a rmi service for this purpose. The rmiUrl parameter contains the rmi location where the external system has to connect. In this rmi service publish the class ServiceActionExecutorResponser with next two methods:

public void returnServiceActionResponse(ServiceActionResponse sar) throws RemoteException;

public void returnServiceActionResponse(ServiceActionResponse[] sars) throws RemoteException;

This ServiceActionExecutorResponser through these methods only redirect or call to the returnServiceActionResponse method of the same ServiceActionExecutor that starts the ServiceAction.

In case the external system does not support rmi protocol the ServiceActionExecutor can implement another type of protocol and the only requirement is to call to the returnServiceActionResponse of the ServiceActionExecutor instance. For example:

```
public void returnServiceActionResponse(ServiceActionResponse sar) throws
RemoteException {
            if (sar == null){
                  log.error("SAEAsyncResponser.returnSAResponse: sar null ",
new SosaException());
                  return;
            }

            ServiceActionExecutor sae =
ServiceActionExecutorsManager.getServiceActionExecutor(
sar.getServiceActionExecutorName());
            if (sae != null){
                  sae.returnServiceActionResponse(sar);
            }else{
                  log.fatal("ServiceActionExecutor '"
+ sar.getServiceActionExecutorName() +"' doesn't exist for ssid '"
 + sar.getSsid() +"'",new SosaException());
            }
      }
```

public void processAsync(String[] ssids, String rmiUrl) : some queues are able to execute ServiceAction in groups. This method will be called in case the queue support and is configured to execute groups of ServiceAction as asynchronous. If the external system doesn't not support groups execution then it's recommended just to make next loop.

```
if (ssids != null)
                  for (int i=0; i<ssids.length;i++)
                        processAsync(ssids[i], rmiUrl);
```

public void returnServiceActionResponse(ServiceActionResponse sar) : when a ServiceAction is called asynchronous and external system return the result this method will be called. As least the method processServiceActionResponse has to be called to tell SOSA the result of the ServiceAction.

```
if(sar != null)
{
      processServiceActionResponse(sar);
}else{
      log.warn("ServiceActionResponse is null");
}
```

public void returnServiceActionResponse(ServiceActionResponse[] sars) : the recommended implementation is next to avoid any errors.

```
public void returnServiceActionResponse(ServiceActionResponse[] sars) throws
RemoteException {
            if (sars != null){
                  for (int i=0; i < sars.length; i++){
                        returnServiceActionResponse(sars[i]);
                  }
            }

      }
```

4 Queues

SOSA provides a method to control the load distribution among the service activation systems, the queues. Queues store Service Actions until a consumer takes them out to be executed by a SAE. Each queue can have several consumers.

Queues are defined in the SOSA configuration file at $SOSA_CONF/`sosa_conf.xml`, under the tag `<Queues>`. Each queue must have its own `<Queue>` tag with the attributes:

`name`: The name of the queue.

`className`: Name of the class that implements the queue.

The rest of the parameters of an implementation of queue are defined as tags `<Parameter>` with attributes "`name`" and "`value`". The concrete implementation of the queue must be responsible of reading the parameters defined.

A queue must have one or more Service Action Executors associated, which are the entities which process the elements of the queue. The SAEs are associated with the queues using the tag `<Sae>` inside the element `<Queue>`. There could be more than one SAE defined for the same queue and the load will be divided into them according to the following tag `<Sae>` attributes:

`name`: The name of the Service Action Executor.

`medium_load`: Average percentage of the queue load the SAE has to support.

Users can build theirs own queues, which manage the Service Action execution as desired. However, SOSA provides a set of implemented queues that are useful most of applications. These are basic FIFO queues, priority queues and dynamic sub-queues.

4.1.1 Basic Queue

This is the basic implementation of a queue, which could be enough for users without any special needs in the queue behaviour. It is implemented as a FIFO (First In First Out) queue with a list where the elements are added when they arrive, and its consumer that reads the first element of the list. The configuration parameters are:

Name	Description
queue.threads	Number of threads consuming from the queue. Default value is one consumer.
queue.scheduler.interval	Milliseconds between two consecutive reads from queue. Default value is 0.
queue.timeout	Maximum time in milliseconds that a Service Action can stay inside the queue. Default value is 0, which means there is no timeout.
queue.max.parallelism	Maximum number of Service Actions being processed

	simultaneously. Default value is 0, which means the max parallelism depends on the SAEs.
queue.block.on.retry	In case the ServiceAction has failed and will be reinserted, the ServiceAction will be reinserted into the first place and the place on maximum parallelism will not be release until the ServiceAction has been reenqueue. By default false.
queue.group	If a queue is defined as a group queue, it processes groups of Service Actions according to the parameters `queue.group.max.time` and `queue.group.max.num`. Default value is `false`.
queue.group.max.time	Maximum time in milliseconds that an element of the group can wait. Default value is 0, which means there is no maximum time.
queue.group.max.num	Number of elements that form a group if `queue.group.max.time` has not expired. Default value is 0, which means there is no maximum number.
queue.wait.retry	Milliseconds that the consumer has to wait before retrying to get an Executor.
queue.synchronous	If it is synchronous, the consumer waits for the service action to finish before getting another one from the queue.

Here is an example configuration of a basic queue with five consumer threads and two MWFM SAEs to which the load is distributed equally:

```
<Queue className="com.hp.sosa.modules.sosamodule.queues.basic.BasicQueue"
name="MWFM_BASIC_QUEUE">
  <Parameter name="queue.threads" value="5"/>
  <Parameter name="queue.synchronous" value="false"/>
  <Sae medium_load="50" name="MWFM_SA_EXECUTOR1"/>
  <Sae medium_load="50" name="MWFM_SA_EXECUTOR2"/>
</Queue>
```

4.1.2 Priority Queue

This implementation of a queue uses the priority of a Service Action to decide which one has to be processed. When a Service Action is assigned to a priority queue in the catalog, it should also include "`priority=X`" in the field `QUEUE_PARAMETERS`. This way, the queue reads this parameter to know the priority of that Service Action.

The algorithm used to consume elements from the queue does not process all the elements of one priority before processing the immediate lower priority. In fact, it is a probabilistic algorithm where the higher the priority an element has, the more probable is for it to be consumed. This way, it is possible that an element of the lowest priority is consumed before an element with the highest one.

By default, there are four priorities defined, from zero to three, where three is the highest, but the number of priorities can be overwritten in the configuration of the queue with the parameter "`priorities`". Each priority defined creates one list where elements of that priority are added.

The configuration parameters of the priority queues are the following:

Name	Description
queue.priorities	Number of priorities which can manage the queue.
queue.threads	Number of threads consuming from the queue. Default value is one consumer.
queue.scheduler.interval	Milliseconds between two consecutive reads from queue. Default value is 0.
queue.timeout	Maximum time in milliseconds that a Service Action can stay inside the queue. Default value is 0, which means there is no timeout.
queue.max.parallelism	Maximum number of Service Actions being processed simultaneously. Default value is 0, which means the max parallelism depends on the SAEs.
queue.block.on.retry	In case the ServiceAction has failed and will be reinserted, the ServiceAction will be reinserted into the first place and the place on maximum parallelism will not be release until the ServiceAction has been reenqueue. By default false.
queue.group	If a queue is defined as a group queue, it processes groups of Service Actions according to the parameters `queue.group.max.time` and `queue.group.max.num`. Default value is `false`.
queue.group.max.time	Maximum time in milliseconds that an element of the group can wait. Default value is 0, which means there is no maximum time.
queue.group.max.num	Number of elements that form a group if `queue.group.max.time` has not expired. Default value is 0, which means there is no maximum number.
queue.wait.retry	Milliseconds that the consumer has to wait before retrying to get an Executor.
queue.synchronous	If it is synchronous, the consumer waits for the service action to finish before getting another one from the queue.

The process of consuming an element from the queue is simple but must be explained:

Decide which priority the consumer is going to use.

If there are elements of this priority, get one of them.

If there are not elements of this priority, check if there are elements of other priorities from highest to lowest, until one element is found.

If there are no elements in the queue, wait until one of any priority arrives and get it.

In case of a grouped queue, the behaviour is quite similar:

Decide which priority the consumer is going to use.

If there are enough elements of this priority, or if there are not enough but any of them have been waiting more than `queue.group.max.time`, get them.

If after the previous step we have not sent any element to be processed, check if there are enough elements of other priorities from highest to lowest, until one with enough elements is found.

If no priority has enough elements to process, check from highest to lowest priority if there are elements in the list that have been waiting more than `queue.group.max.time`, and send them if any is found.

If no element has been sent yet, wait until another element is added, and check from highest to lowest priority if a group is formed or if timeout is finished.

Here is an example configuration of a priority queue with five consumer threads and two MWFM SAEs to which the load is distributed equally:

```
<Queue className=" com.hp.sosa.modules.sosamodule.queues.priority.PriorityQueue"
name="MWFM_PRIORITY_QUEUE">
  <Parameter name="queue.threads" value="5"/>
  <Parameter name="queue.synchronous" value="false"/>
  <Parameter name="queue.priorities" value="4"/>
  <Parameter name="queue.group.max.num" value="10"/>
  <Parameter name="queue.group.max.time" value="3000"/>
  <Sae medium_load="50" name="MWFM_SA_EXECUTOR1"/>
  <Sae medium_load="50" name="MWFM_SA_EXECUTOR2"/>
</Queue>
```

4.1.3 Dynamic Basic Subqueue

This is the basic implementation of a queue with subqueues. The queue would get the “subque name” and if this subqueue is not defined it’s created dynamically. In each subqueue it’s implemented as a FIFO (First In First Out) queue with a list where the elements are added when they arrive, and its consumer that reads the first element of the list. The consumers will find the next subqueue available with elements to execute. There are two level of maximum of parallelism, the global maximum elements to execute and the max parallelism for each subqueue. Also, the subqueues are available to lock/unlock and open/close. If queue is locked it’s locked for all queue. Then a subqueue is unlocked if que queue is unlocked and subqueue unlocked. Same to open/close. The group execution is not supported in this type of queue.

To get the subqueue name a class have to be defined. The default class is com.hp.sosa.modules.sosamodule.queues.dynamicsubqueue.SubQueueGetter. If you want to create another you have to extends this class an overwrite the method public String getSubQueue(String ssid). If this class return a empty subqueue name, the service will add to default suqueue. It’s possible to define the extended class into the catalog on subqueue assign class field.

The configuration parameters are:

Name	Description
queue.threads	Number of threads consuming from the queue. Default value is one consumer.
queue.scheduler.interval	Milliseconds between two consecutive reads from queue. Default value is 0.
queue.timeout	Maximum time in milliseconds that a Service Action can stay inside the queue. Default value is 0, which means there is no timeout.
queue.max.parallelism	Maximum number of Service Actions being processed simultaneously. Default value is 0, which means the max parallelism depends on the SAEs.
queue.wait.retry	Milliseconds the consumer has to wait before retrying to get an Executor.
queue.synchronous	If it is synchronous, the consumer waits for the service action to finish before getting another one from the queue.
queue.block.on.retry	In case the ServiceAction has failed and will be reinserted, the ServiceAction will be reinserted into the first place and the place on maximum parallelism will not be release until the ServiceAction has been reenqueue. By default false.
queue.subqueue.max.parallelism	Default sub-queue max parallelism.
queue.subqueue.getter.class	The sub-queue getter name class (default class will find `subqueue.name` parameter first in `QUEUE_PARAMETERS` and then in `InputParams`).
queue.subqueue.max.time.live	The max time that a sub-queue not defined can live (default 0, which implies infinite).
queue.subqueue.name0	Name of sub-queue 0.
queue.subqueue.max.parallelism0	Maximum parallelism to sub-queue 0.
queue.subqueue.name1	Name of sub-queue 1.
queue.subqueue.max.parallelism1	Maximum parallelism to sub-queue 1.
…	(You can define all sub-queues that you want).

4.1.4 Dynamic Priority Subqueue

This is the priorty implementation of a queue with subqueues. The queue would get the “subque name” and if this subqueue is not defined it’s created dynamically. In each subqueue it’s implemented as a

FIFO (First In First Out) queue with a list where the elements are added when they arrive, and its consumer that reads the first element of the list. The consumers will find the next subqueue available with elements to execute. There are two level of maximum of parallelism, the global maximum elements to execute and the max parallelism for each subqueue. Also, the subqueues are available to lock/unlock and open/close. If queue is locked it's locked for all queue. Then a subqueue is unlocked if que queue is unlocked and subqueue unlocked. Same to open/close. The group execution is not supported in this type of queue.

To get the priority this queue uses the same system as "Priority Queue".

To get the subqueue name a class have to be defined. The default class is com.hp.sosa.modules.sosamodule.queues.dynamicsubqueue.SubQueueGetter. If you want to create another you have to extends this class an overwrite the method public String getSubQueue(String ssid). If this class return a empty subqueue name, the service will add to default suqueue. It's possible to define the extended class into the catalog on subqueue assign class field.

The configuration parameters are:

Name	Description
queue.threads	Number of threads consuming from the queue. Default value is one consumer.
queue.scheduler.interval	Milliseconds between two consecutive reads from queue. Default value is 0.
queue.timeout	Maximum time in milliseconds that a Service Action can stay inside the queue. Default value is 0, which means there is no timeout.
queue.max.parallelism	Maximum number of Service Actions being processed simultaneously. Default value is 0, which means the max parallelism depends on the SAEs.
queue.wait.retry	Milliseconds the consumer has to wait before retrying to get an Executor.
queue.synchronous	If it is synchronous, the consumer waits for the service action to finish before getting another one from the queue.
queue.block.on.retry	In case the ServiceAction has failed and will be reinserted, the ServiceAction will be reinserted into the first place and the place on maximum parallelism will not be release until the ServiceAction has been reenqueue. By default false.
queue.subqueue.max.parallelism	Default sub-queue max parallelism.
queue.subqueue.getter.class	The sub-queue getter name class (default class will find `subqueue.name` parameter first in `QUEUE_PARAMETERS` and then in `InputParams`).
queue.subqueue.max.time.live	The max time that a sub-queue not defined can live

	(default 0, which implies infinite).
queue.subqueue.name0	Name of sub-queue 0.
queue.subqueue.max.parallelism0	Maximum parallelism to sub-queue 0.
queue.subqueue.name1	Name of sub-queue 1.
queue.subqueue.max.parallelism1	Maximum parallelism to sub-queue 1.
…	(You can define all sub-queues that you want).

4.1.5 ServiceActionExecutor Selector

By default all queues use the ServiceActionExecutorSelector class to decide which ServiceActionExecutor will be used to execute the ServiceAction. It's possible to extends and use a own implemented class. For this purpose the class com.hp.sosa.modules.sosamodule.queues. ServiceActionExecutorSelector needs to be extended. Let's suppose we call to this class NewSelector

```
public class NewSelector  extends ServiceActionExecutorSelector
```

In other hand, the into the queue section of sosa_conf.xml this selector needs to be configured.

```
<Queue className="com.hp.sosa.modules.sosamodule.queues.basic.BasicQueue"
name="MWFM_BASIC_QUEUE" saeSelectorClassName=”mypackage.NewSelector”>
  <Parameter name="queue.threads" value="5"/>
  <Parameter name="queue.synchronous" value="false"/>
  <Sae medium_load="50" name="MWFM_SA_EXECUTOR1"/>
  <Sae medium_load="50" name="MWFM_SA_EXECUTOR2"/>
</Queue>
```

public void init(Sae[] saes, Map parameters) : initialize the selector

public ServiceActionExecutor getServiceActionExecutor() : return the next serviceActionExecutor to use

public synchronized void addServiceActionExecutor(Sae sae) : add new ServiceActionExecutor to this selector

public synchronized void removeServiceActionExecutor(String name) : remove new ServiceActionExecutor to this selector

public Sae[] getSaes() : get the current ServiceActionExecutors

5 Service Catalog

The SOSA Service Catalog basically consists on the list of services provided by SOSA to the clients.

5.1.1 Definition

When SOSA receives a Service Order to process, it first needs to build it according to its catalog of services. The structure of a single Service Order in the catalog is a tree with a root node and any number of branches of unlimited depth with more nodes as children. There are two types of nodes in the tree: Service Orders and Service Actions.

A Service Order is a node which can have other Service Orders or Service Actions as children, while a Service Action cannot have any children as it is a single executable action.

The following figure shows an example of a tree service:

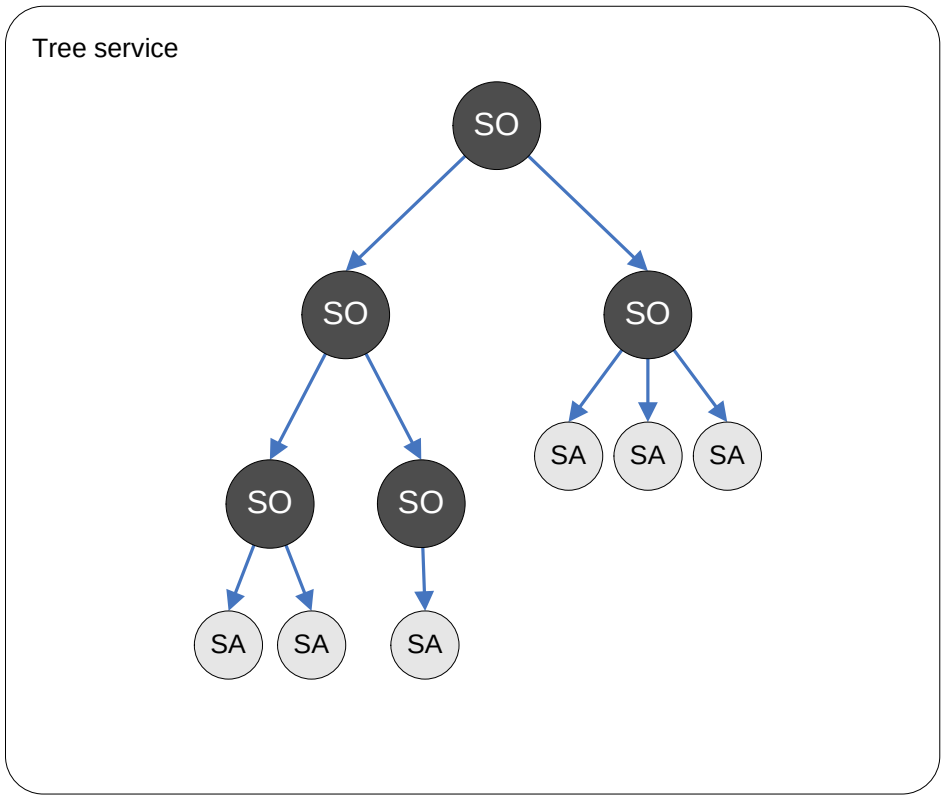

Fig. 2: Tree service example

In this example we can see that this service order is compound of two service orders in the first level of depth; one of them is another complex service order, and the other is a single service order with three service actions to execute.

With this structure in the catalog, the developer could design service orders as complex as needed, because there is no limit in the number of children a service order can have, and so, no limit in the number of service actions that can be executed with a simple service order.

Another important feature derived from this structure is the flexibility to execute different orders or actions in serial or parallel mode. The processing mode is defined in the service order, which means that a service order is defined to process its children in serial or parallel mode. But as a service order can have another service order as a child, there could be different modes defined on each branch of the tree. For example, the root could execute its children in parallel mode, and each one of those could execute their children in serial or parallel too, and so on with deeper nodes. Service orders and actions can also be marked to be executed offline, meaning that they will immediately return the OK status to their parent service order and continue executing in the background.

5.1.2 JDBC Service Catalog

With the basic installation of SOSA, there is a catalog implementation included which stores all the elements into an Oracle database, and uses a JDBC driver to connect to it.

The configuration of the catalog is included in the SOSA configuration file at `$SOSA_CONF/sosa_conf.xml`. To define a catalog implementation there must be three attributes into the `<Catalog>` tag, which are:

`type`: Must be "`catalog`".

`name`: Name of the catalog.

`className`: Class name that implements the catalog.

The JDBC catalog supports several configuration parameters:

Name	Description
catalog.db.pool.name*	Name of the database connections pool.
catalog.db.host*	Hostname of the machine where the database is present.
catalog.db.port*	Port to connect.
catalog.db.instance*	Oracle instance which stores the catalog.
catalog.db.user*	Username of the instance used for the catalog.
catalog.db.password*	Password of the username.
catalog.db.driver.name*	Name of the driver used.
catalog.db.jdbc.driver*	JDBC driver used.
catalog.db.initialsize	Initial size of connections pool.
catalog.db.maxactive	Maximum time a connection is active.
catalog.db.maxidle	Maximum time a connection is idle.
catalog.db.minidle	Minimum time for a connection to be marked as idle.

catalog.db.maxwait	Maximum time waiting for a connection.

Those parameters followed by an asterisk(*) are mandatory, so in case one of them is not present in the configuration file an exception is launched in the catalog initialization.

Here is an example of a JDBC catalog configuration:

```
<Catalog className="com.hp.sosa.modules.sosamodule.catalog.jdbc.JDBCTreeCatalog"
name="JDBC_Catalog" type="catalog">
  <Parameter name="catalog.db.reload.period" value="600000"/>
  <Parameter name="catalog.db.autoreload" value="true"/>
  <Parameter name="catalog.db.pool.name" value="db_sosa_catalog"/>
  <Parameter name="catalog.db.user" value="userpp"/>
  <Parameter name="catalog.db.password" value="userpp"/>
  <Parameter name="catalog.db.jdbc.driver" value="oracle.jdbc.driver.OracleDriver"/>
  <Parameter name="catalog.db.driver.name" value="jdbc:oracle:thin"/>
  <Parameter name="catalog.db.host" value="127.0.0.1"/>
  <Parameter name="catalog.db.port" value="1521"/>
  <Parameter name="catalog.db.instance" value="HPSA"/>
</Catalog>
```

5.1.3 Building Service Catalog

The usual process to define a service catalog is composed of three phases:

Define the Service Actions.

Define the relationships between Service Orders and Service Actions.

Define the relationships between Service Orders.

When Service Orders and Service Actions are defined several properties have to be set. These properties will state the operation of the services. Next sections will show the configuration options in the service catalog.

5.1.3.1 Service Orders

Service Orders configuration parameters are the following:

Name	Description
service_order_name	Name of the service order.
service_name	Name of the service.
operation	Operation to execute.
persistable	A Boolean to specify if the service order must be persisted or not. If a Service Order is marked as persistent, the tree of services built from the Service Order will be storage in the persistent data base during its

	execution.
timeout	The time in milliseconds before a timeout occurs.
processing_mode	Children will be executed in serial or parallel mode. If `serial`, the Service Order children will be executed one by one following the order assigned in the catalog; if `parallel`, all children will be executed at the same time.
state	State of the service order: `enable`, `disable`, `reject` or `simulate`.
on_error	If there is an error on this Service Order, the action it will execute (`ABORT`, `SUSPEND`, `ROLLBACK` or `CONTINUE`).
offline	The service order will return immediately to its parent (or the protocol adapter if it is the order root) with an OK code and continue executing as usual afterwards. Actual return code and description can be checked in the history or persistence views.

All three parameters, "service_order_name", "service_name", and "operation" are the Service Order identifier.

5.1.3.2 Service Actions

Service Orders configuration parameters are the following:

Name	Description
service_action_name	Name of the service action.
service_name	Name of the service.
operation	Operation to execute.
wf_name	Workflow name.
wf_name_undo	Workflow name in undo.
extra_info	Extra info for the processor.
work_queue	Name of the queue that process the Service Action.
state	State of the service order: `enable`, `disable`, `reject`, `simulate`.
num_retry	Number of retries in case of error.
retry_interval	Miliseconds between two consecutive retries.
user_mapper_class	Class used to map parameters among Service Actions of the same Service Order.
timeout	The time in milliseconds before a timeout occurs.
subqueue_parameter	Name of the parameter used to assign the sub-queue.
subqueue_assign_class	Name of the class used to assign the sub-queue.

error_codes_retry	Error codes list that make the Service Action to be retried.
num_retry_list	Number of retries for each error code.
end_action_class	User class that manages the end of a Service Action.
on_error_wait_operator	On error, a manual action should be done.
queue_parameters	Parameters used by the queue in the form param1=value1, param2=value2…
use_sa_unique_name	Use the specified service action name instead of the parent service order name. This option is provided to preserve compatibility with previous versions of SOSA where service actions had no service_action_name parameter.
offline	The service order will return immediately to its parent (or the protocol adapter if it is the order root) with an OK code and continue executing as usual afterwards. Actual return code and description can be checked in the history or persistence views.

The parameters “service_action_name”, “service_name” and “operation” are the Service Action identifier.

5.1.3.3 Parameters Mapping

Parameters mapping are associated to a Service Action. It allows making variable copies among different or same variable spaces. A variable space is a context where a set of variables may exist.

SOSA defines the following variable spaces:

Input space: Variable space that contains the input variables of a Service Order.

Output space: Variable space that contains the output variables of a Service Order.

Context space: Variable space that contains the context variables of a Service Order.

Results space: Variable space that contains the result variables of a Service Action execution.

There are four kinds of parameter mappings:

Input Mappings DO: These mappings are executed at the beginning of the Service Action execution.

Input Mappings UNDO: These mappings are executed at the beginning of the rollback Service Action execution.

Output Mappings DO: These mappings are executed at the end of the Service Action execution.

Output Mappings UNDO: These mappings are executed at the end of the rollback Service Action execution.

Within each kind of mapping, several mappings are ordered, so that they are executed in the established order.

5.1.3.4 Service Parameters

A collection of service parameters can be defined in the catalog for a service action. Service parameters allow validating the request input parameters for the service action and providing default values for them. To preserve backwards compatibility, if no service parameters are specified for a service action or any parameter present in the request is not defined as a service parameter for the service action, it will be passed to the executor as-is.

Service parameters are defined by:

Name	Description
name	Name of the service parameter.
type	Data type of the service parameter. Allowed types are String, Integer, Long, Boolean and Date.
format	Parameter format for validation purposes. String parameter format is defined by a regular expression. Boolean parameters have no format. For Integer, Long and Date parameters a maximum and/or minimum value can be specified. Optional.
default value	The parameter default value. Optional.
mandatory	Indicates that the parameter must be defined in the request.
overwrite	Indicates that the default value takes precedence over the request value.
enabled	If not enabled, the parameter is omitted during validation and its default value won't overwrite the request value in any case.

5.1.4 Dynamic Service Orders

Dynamic service orders allow creating service orders at request time by combining catalog service orders and actions and providing parameters to them. This special kind of request is sent to the protocol adapter in XML format.

5.1.4.1 Request XML Schema

The dynamic order request must adhere to the corresponding XML schema. The dynamic order schema can be found in `$SOSA_CONFIG/xsd/dso.xsd`.

5.1.4.2 Request Semantics

A dynamic order request consists of the following elements:

a. serviceRequest

This is the root element. It must contain a <services> child element as well as it may have an optional <header> element. It has no attributes.

b. header

This is a non mandatory child of the <serviceRequest> element. Consists of one or more <param> elements with mandatory <name> and <value> children for specifying input parameters common to all services in the request.

c. services

This is a mandatory child of the <serviceRequest> element which acts as a container for a service or group of services. It will be translated to a dynamic service order at build time. It supports the following attributes:

- mode: possible values are “serial” or “parallel”. It specifies how the children services have to be executed. Same as the processing_mode parameter of a catalog service order.
- onerror: possible values are “abort”, “suspend”, “rollback” or “continue”. It specifies what to do when a children service returns an error code. Same as the on_error parameter of a catalog service order.
- persistence: possible values are “enable” or “disable”. It specifies if the contained services must be persisted, same as the persistable attribute of a catalog service order. All services inherit their parent persistence setting at order build time so only the root <services> setting will be taken into account.
- name, type, action: the identifiers for the dynamic service order that will be created.
- scheduledStartTime: the time at which the service order should start to be processed.

A <services> element must contain one or more <service> or <services> elements defining the children services.

d. service

This is a mandatory child of the <services> element defining a specific service. The following attributes are supported:

- name, type, action: those three mandatory children elements identify a specific catalog service order or action to be instantiated.
- characteristics: similar to the <header> element, it consists of one or more <characteristic> elements with their corresponding <name> and <value> children specifying additional input parameters for the parent service. If a certain characteristic name is also a header parameter name, the characteristic value takes precedence when creating the service input parameters map. This is an optional child element.
- composite: this is an optional child element that contains a <services> element representing the dependencies of the parent service. If it is present the service order resulting from it will be executed after the service by creating a dynamic service order with them both in serial mode.
- scheduledStartTime: the time at which the service should start to be processed.

NOTE: In cases where a dynamic service order would contain a single service order or action (except in the case it is the root of the order tree) it will be omitted in order to reduce the number of services in the order tree, so that single service child would directly replace the parent unnecessary dynamic order.

5.1.4.3 Service Request Example

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<serviceRequest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.hp.com/sosa/dynamicserviceorder dso.xsd"
    xmlns="http://www.hp.com/sosa/dynamicserviceorder">

    <header>
        <param>
            <name>customerid</name>
            <value>34892</value>
        </param>
        <param>
            <name>systemrequestorid</name>
            <value>OrderManagement</value>
        </param>
    </header>

    <services mode="serial" type="rollback" persistence="enable">

        <service scheduledStartTime="2011-11-10T18:23:00.000+01:00">
            <name>VOICE</name>
            <type>MOBILE</type>
            <action>ACTIVATE</action>
            <characteristics>
                <characteristic>
                    <name>imsi</name>
                    <value>1234124312342352133</value>
                </characteristic>
                <characteristic>
                    <name>msisdn</name>
                    <value>34979646947</value>
                </characteristic>
                <characteristic>
                    <name>rsa</name>
                    <value>42</value>
                </characteristic>
            </characteristics>
            <composite>
                <services mode="parallel" type="continue">
                    <service>
                        <name>VMS</name>
                        <type>MOBILE</type>
```

```
                    <action>ACTIVATE</action>
                    <characteristics>
                        <characteristic>
                            <name>imsi</name>
                            <value>1234124312342352133</value>
                        </characteristic>
                        <characteristic>
                            <name>msisdn</name>
                            <value>34979646947</value>
                        </characteristic>
                        <characteristic>
                            <name>capacity</name>
                            <value>5Mb</value>
                        </characteristic>
                    </characteristics>
                </service>
            </services>
        </composite>
    </service>

    <service>
        <name>DATA</name>
        <type>MOBILE</type>
        <action>ACTIVATE</action>
        <characteristics>
            <characteristic>
                <name>imsi</name>
                <value>1234124312342352133</value>
            </characteristic>
            <characteristic>
                <name>msisdn</name>
                <value>34979646947</value>
            </characteristic>
            <characteristic>
                <name>capacity</name>
                <value>5Mb</value>
            </characteristic>
        </characteristics>
    </service>
```

```
    </services>


</serviceRequest>
```

6 Protocol Adapters

In order to receive incoming requests from several clients using different protocols, SOSA define the Protocol Adapter concept.

6.1.1 Definition

Protocol Adapters are the components which listen to and handle the queries sent by the external clients and build the tree service that will be inserted into SOSA.

Developers can extend an abstract class called "`protocoladapter`" to create custom protocol adapters able to manage to the communications between SOSA and the client side.

To use a protocol adapter, it must be declared in the SOSA configuration file at `$SOSA_CONF/sosa_conf.xml`. Protocol adapters are defined within a `<ProtocolAdapter>` tag inside the `<ProtocolAdapters>` tag. Each `<ProtocolAdapter>` tag must contain two attributes:

`name`: Name of the protocol adapter.

`className`: Class name that implements the protocol adapter.

Although SOSA allows building customized protocol adapters, it also provides two implemented protocol adapters to work with RMI and Web Services clients.

6.1.2 RMI and Web Service Protocol Adapters

Both of the protocol adapters, RMI and Web Service, can work in two ways synchronous and asynchronous mode. Into the asynchronous mode it can work as pure asynchronous communication or pooling mode.

Synchronous mode: after receiving a request, the protocol adapter builds and inserts the service tree into SOSA, then wait for the response to notify the client.

Pure asynchronous mode: in this mode, the protocol adapter does not wait for the response. The protocol adapter will return to the clien, this means the client should publish the rmi or webservice interface to get the results.

Pooling asynchronous mode: in this mode, the protocol adapter does not wait for the response. It provides methods to check if a service order is processed and to get the response.

From the standpoint of performance the asynchronous mode is usually the best choice. This mode allows clients implement pulling methods to handle the request, that are more efficient in managing threads that synchronous mode.

e. RMI Protocol Adapter

This protocol adapter is defined in the SOSA configuration file as follows:

```
<ProtocolAdapters>
  <ProtocolAdapter
className="com.hp.sosa.modules.sosamodule.protocoladapters.rmi.RMIProtocolAdapter"
name="RMI_PA">
    <Parameter name="rmi.service.name" value="RmiPA"/>
    <Parameter name="pooling.mode" value="false"/>
    <Parameter name=" rmi.response.url " value="//ip:port/servicename"/>
  </ProtocolAdapter>
</ProtocolAdapters>
```

The protocol adapter has next configurable parameters:

Name	Description
rmi.service.name	name of the RMI service name with which the protocol adapter will be registered via RMI
rmi.response.url	The client rmi url where the protocol adapter will return the response. The service should implement RMIPAResponseClient class
pooling.mode	If true the answer need to be getted by the client, if false the answer will be sent to the response url service.

SOSA always use the same host and port to register all RMI protocol adapters:

Host: Host where SOSA is running.

Port: 1119

f. Web Service Protocol Adapter

Below is shown an example configuration of Web Service protocol adapter:

```
<ProtocolAdapters>
  <ProtocolAdapter
className="com.hp.sosa.modules.sosamodule.protocoladapters.ws.WSProtocolAdapter"
name="WS_PA">
    <Parameter name="ws.ip" value="127.0.0.1"/>
    <Parameter name="ws.port" value="8070"/>
    <Parameter name="ws.min.threads" value="2"/>
    <Parameter name="ws.max.threads" value="10"/>
    <Parameter name="ws.url" value="/axis"/>
  </ProtocolAdapter>
</ProtocolAdapters>
```

The protocol adapter has the following parameters:

Name	Description
ws.ip	The Web Service IP.
ws.port	The Web Service port.
ws.min.threads	The minimum number of threads serving requests.
ws.max.threads	The maximum number of threads serving request.
ws.url	The Web Service URL.
ws.webapp.path	The Web Service app path (by default a ./webapps/axis1.4)
ws.response.url	The response url web services. The web service needs to implement the attached wsdl
pooling.mode	If true the answer need to be getted by the client, if false the answer will be sent to the response url service.

g. Next Generation Web Service Protocol Adapter

The Next Generation Web Service (NGWS) protocol adapter is a reimplementation of the Web Service protocol adapter using newer technologies such as JAXB and JAX-WS. This leads to a cleaner and more compatible web service interface for SOSA. Below is shown an example configuration of the NGWS protocol adapter:

```
<ProtocolAdapter
className="com.hp.sosa.modules.sosamodule.protocoladapters.ngws.NGWSProtocolAdapter"
name="NGWS_PA">
       <Parameter name="ngws.host" value="127.0.0.1"/>
       <Parameter name="ngws.port" value="8070"/>
       <Parameter name="ngws.min.threads" value="2"/>
       <Parameter name="ngws.max.threads" value="10"/>
       <Parameter name="ngws.path" value="ngws"/>
</ProtocolAdapter>
```

The protocol adapter has the following parameters:

Name	Description
ngws.host	Server listening hostname.
ngws.port	Server listening port.
ngws.min.threads	The minimum number of threads serving requests.
ngws.max.threads	The maximum number of threads serving request.
ngws.path	The URL context path.
ngws.webapp	The webapp directory location. Optional.
pooling.mode	If true, responses to asynchronous requests must be retrieved by the client, otherwise they will be sent to the

	service specified by ngws.response.url.
ngws.response.url	Response URL for asynchronous requests when pooling.mode is inactive.

6.1.3 Implementing the client side

With both protocol adapters, RMI and Web Services, a basic java client is delivered. These clients manage the connections, relieving the customer to its management, and provide methods to send and manage requests.

Both of clients (`RMIPAClientService` and `WSPAClientService`) offer commons methods to work with service orders:

`getInstance()`: gets an instance of the client.

`startServiceOrderAsync(type, service, action, inputParams, user)`: starts a service order identified by type, service and action, with the given input parameters and the user who send it.

`startDynamicOrderAsync(request, user)`: starts a dyanmic order based on the given XML formatted request and the user who send it in an asynchronous manner.

`startDynamicOrderSync(request, user)`: starts a dyanmic order based on the given XML formatted request and the user who send it in a synchronous manner.

`isServiceOrderReturned(ssid)`: checks if a service order identified by ssid has finished its execution.

`getReturnedServiceOrder(ssid)`: gets the finished service order identified by ssid.

In case pooling.mode is false the rmi client should implement a rmi service base on RMIPAResponseClient interface class. In case of webservice the client must implement next wsdl.

WSPAReturnService.wsdl

Using the Protocol Adapter clients, customers can develop theirs owns clients by using these methods.

6.1.4 Creation new Protocol Adapter

Usually is required to create new protocol adapter to adapt the external system format or protocol to SOSA. For this purpose we create new protocol adapter implementations. Let's suppose the new protocol adapter is name NewExampleProtocolAdapter. The first step is extend the class com.hp.sosa.modules.sosamodule.protocoladapters.ProtocolAdapter.

```
public class NewExampleProtocolAdapter extends ProtocolAdapter {
```

After this step there're different methods required to implement.

public void finish() : when SOSA is stopping this method is called by SOSA. Here is required any step require to unconfigure the protocol adapter or finish the connections.

public String getErrorMessage() : in case the protocol adapter detect some error with the external system return a error message. Leave empty if not possible to detect any errors.

```
	public String getErrorMessage() {
		return null;
	}
```

public String getGlobalStatus() : return the status of the protocol adapter

```
	public String getGlobalStatus() {
		if (isRunning()){
			return Constants.SOSA_ACTION_RESUME;
		}else{
			return Constants.SOSA_ACTION_PAUSE;
		}
	}
```

public String[] getListenersName() : in some case is necessary to add listeners to the protocol adapter. For example, if we define a jdbc protocol adapter and we can connect to two databse it's possible to define two listener, one for each database. Usually, is more common to define more instance of protocol adapter to avoid using listeners. This method return the name of listeners. Retun {} is there aren't.

```
	public String[] getListenersName() {
		String[] listeners = {};
		return listeners;
	}
```

public boolean haveToInsertService(String ssid) : when SOSA start and detects the root ServiceOrder is on BUILDED status means that the protocol adapter made the build of the order only request to insert into the core. In the protocol adapter want to insert the order in SOSA needs to return

true, if not return false. Usually, we return false because probably the external system couldn't receive the id.

```
public boolean haveToInsertService(String ssid) {
	return false;
}
```

public void init(Map initParameters) : the initParameters map contains all the parameters configured into the sosa_conf.xml file for this ServiceActionExecutor. This method will be called by SOSA when start and before to send any ServiceAction to the executor to be able to configure it.

public void refreshConfiguration(Map initparms) : the initParameters map contains all the parameters configured into the sosa_conf.xml file for this ServiceActionExecutor. This method will be called by SOSA when the configuration needs to be refreshed. In case is not possible to refresh the configuration leave empty this method.

public boolean isRunning() : return true if the protocol adapter is running. Return false in other case.

```
public boolean isRunning() {
	return running;
}
```

public boolean isRunningListener(String listenerName) : return true if the listener is running. Return false in other case.

```
public boolean isRunningListener(String name) {
	for (int i = 0; i < listeners.length; i++) {
		if (name.equals(listeners[i])) {
			return listenersRunning[i];
		}
	}
	return false;
}
```

public boolean[] isRunningListeners() : return the listener running status

```
public boolean[] isRunningListeners() {
	return listenersRunning;
}
```

public void startListener(String listenerName) : stop the listener. In case there's not listeners leave empty.

public void stopListener(String listenerName) : star the listener. In case there's not listeners leave empty.

public void pause() : pause the protocol adapter. In other hand, in the method that you use to insert orders is necessary to check the status of protocol adapter to avoid insert orders when is paused.

```
	public void pause() {
		running = false;
	}
```

public void resume() : resume the protocol adapter

```
	public void resume() {
		running = true;
	}
```

public void returnServiceOrder(String ssid) : this is one of the most important method of protocol adapter because when SOSA ends an order call this method and this method has to return the result to the external system. In case, the external system is down or there's a communication problem to not send the result, a SosaException has to be throw. Automatically SOSA will retry after a while when receive this exception.

```
	public void returnServiceOrder(String ssid) throws SosaException {
		ServiceOrder so =
ServiceElementsManager.getServiceOrderReadOnly(ssid);
	       if (so == null) return;
	       try {
	         //returning the order to the external system

	        } catch (Exception e) {
	             new SosaException(e);
	        }
	}
```

public void shutdown() : When SOSA is stopping this method will be called and then the protocol adapter can disconnect or un-configure the required variables or environment

Finally, the protocol adapter needs to insert orders into SOSA. There's no require to implement any particular method but the protocol adapter will need to implement some method that the external system will call. Let's suppose this method is called startOrder:

```
      public String startOrder(String type, String service, String action,
Map inputParams, String user) throws SosaException {
            if (this.isRunning()) {
                  log.debug("starting asynchronous service order; type: " +
type + ", service: " + service + ", action: " + action);
                  String ssid = super.getNewBuildedServiceOrder(type,
service, action, inputParams, user);
                  super.insertServiceOrder(ssid);
                  return ssid;
            }

            log.info("it was not possible to start asynchronous service
order, the protocol adapter is not running");
            throw new SosaException("cannot start the service order, the
protocol adapter is not running");
      }
```

There's two important points to do into this method:

Create the order using the catalog. For this purpose the super class protocol adapter provides a lot of method with names like getNewBuildedServicerOrder.

Insert into SOSA : it's mandatory to insert the order when is ready to process calling the method insertServiceOrder(ssid);

7 Persistence

SOSA3 is provided with a method to let users and developers to configure the persistence package that allows to Sosa keeping a non-volatile register of Service Orders and Service Actions being managed, in order to prevent the loss of received petitions due to falling of the system. And providing the capacity to restore these jobs at the point they where when the fall of the system happened.

The persistence package to use may be configured via the SOSA configuration file, at `$SOSA_CONF/sosa_conf.xml`, by means of the `<Persistence>` tag, where you can specify the main class of the persistence implementation and the configuration parameters it needs.

Developers of a new persistence method have to extend the `SosaPersistence` abstract class and implement its methods.

7.1 Hibernate Persistence

SOSA3 is provided with a persistence based on Hibernate (an object/relational mapping tool). Hibernate is an object/relational mapping tool for Java environments. The term object/relational mapping (ORM) refers to the technique of mapping a data representation from an object model to a relational data model with a SQL-based schema.

Hibernate is the layer in the application which connects to the database, so it needs connection information. The connections are made through a JDBC connection pool, which it also needs to be configured.

To configure Hibernate SOSA uses the XML configuration file, at `$SOSA_CONF/hbm_persistence.cfg.xml`. Below is an example of it:

```
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">
      oracle.jdbc.driver.OracleDriver
    </property>
    <property name="hibernate.connection.url">
      jdbc:oracle:thin:@127.0.0.1:1521:HPSA
    </property>
    <property name="hibernate.connection.username">userpp</property>
    <property name="hibernate.connection.password">userpp</property>

    <!-- C3P0 connection pool -->
    <property name="hibernate.c3p0.acquire_increment">1</property>
    <property name="hibernate.c3p0.idle_test_period">60</property> <!-- seconds -->
    <property name="hibernate.c3p0.max_size">20</property>
    <property name="hibernate.c3p0.max_statements">0</property>
    <property name="hibernate.c3p0.min_size">10</property>
    <property name="hibernate.c3p0.timeout">60</property> <!-- seconds -->

    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.Oracle10gDialect</property>

    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>
    <property name="transaction.factory_class">
```

```
      org.hibernate.transaction.JDBCTransactionFactory
    </property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">false</property>

    <!-- Disable the second-level cache  -->
    <property name="hibernate.cache.provider_class">
      org.hibernate.cache.NoCacheProvider
    </property>

    <!-- Possible values are: create(always create a new schema when starts),
    update(updatetables that are different) or comment next property to do nothing
    -->
    <property name="hibernate.hbm2ddl.auto">update</property>

    <mapping resource="hibernate/ServiceOrder.hbm.xml"/>
    <mapping resource="hibernate/ServiceAction.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

The first four property elements contain the necessary configuration for the JDBC connection. After, it is the configuration of the C3P0 connection pooling tool. The dialect property element specifies the particular SQL variant Hibernate generates. Hibernate's automatic session management for persistence contexts is configured to use the Hibernate built in SessionFactory and the thread execution to track the current session. The `hbm2ddl.auto` option turns on automatic updating of database schemas. Finally, we add the mapping files for persistent classes to the configuration – these are `ServiceOrder.hmb.xml` and `ServiceAction.hbm.xml`, at $SOSA_CONF/`hibernate` directory.

Hibernate needs to know how to load and store objects of the persistent class. This is where the Hibernate mapping file comes into play. The mapping file tells Hibernate what table in the database it has to access, and what columns in that table it should use.

SOSA needs to know the configuration of persistence by means of its configuration file at $SOSA_CONF/`sosa_conf.xml`. Below is a configuration example:

```
<Persistence
className="com.hp.sosa.modules.sosamodule.persistence.hibernate.HibernateSosaPersistence"
name="PM" type="persistence">
  <Parameter name="persistence.max.cache.size" value="1000"/>
  <Parameter name="persistence.time.max.in.cache" value="30000"/>
  <Parameter name="persistence.hibernate.config.file" value="hbm_persistence.cfg.xml"/>
</Persistence>
```

The first two parameters contain the configuration for the cache:

`persistence.max.cache.size`: the maximum number of elements that can be kept in cache. This parameter should be high enough to keep a big number of elements in cache but not too high to keep an efficient performance, so it depends on the machine where SOSA is installed. Values higher than 1500 are not recommended.

`persistence.time.max.in.cache`: and the maximum time (expressed in milliseconds) that each element can remain in cache before being removed.

The following parameter:

`persistence.hibernate.config.file`: indicates the Hibernate configuration file that should be loaded to initialize Hibernate.

In this persistence is configured into SOSA then the file $JBOSS_HOME/server/diagnostic/deploy/hpovact.sar/activator.war/properties/hbm_persistence.properties needs to have next content

```
persistence.hibernate.config.file =
/etc/opt/OV/ServiceActivator/config/sosa/hbm_persistence.cfg.xml
```

7.2 File mixed Hibernate Persistence

SOSA3 is provided with a persistence for high performance solutions. This persistence first saved the order in file because is much faster and in case the order spendsa lot of time inside of SOSA change the persistence to hibernate. While the order is save into the file is not possible to search into the administration web.

To configure Hibernate on this persisntece SOSA uses the XML configuration file, at $SOSA_CONF/`hbm_mixedpersistence.cfg.xml`. Below is an example of it:

```
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">
      oracle.jdbc.driver.OracleDriver
    </property>
    <property name="hibernate.connection.url">
      jdbc:oracle:thin:@127.0.0.1:1521:HPSA
    </property>
    <property name="hibernate.connection.username">userpp</property>
    <property name="hibernate.connection.password">userpp</property>

    <!-- C3P0 connection pool -->
    <property name="hibernate.c3p0.acquire_increment">1</property>
    <property name="hibernate.c3p0.idle_test_period">60</property> <!-- seconds -->
    <property name="hibernate.c3p0.max_size">20</property>
    <property name="hibernate.c3p0.max_statements">0</property>
    <property name="hibernate.c3p0.min_size">10</property>
    <property name="hibernate.c3p0.timeout">60</property> <!-- seconds -->

    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.Oracle10gDialect</property>

    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>
    <property name="transaction.factory_class">
      org.hibernate.transaction.JDBCTransactionFactory
```

```
    </property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">false</property>

    <!-- Disable the second-level cache  -->
    <property name="hibernate.cache.provider_class">
      org.hibernate.cache.NoCacheProvider
    </property>

    <!-- Possible values are: create(always create a new schema when starts),
    update(updatetables that are different) or comment next property to do nothing
    -->
    <property name="hibernate.hbm2ddl.auto">update</property>

    <mapping resource="hibernate/MixedServiceOrder.hbm.xml"/>
    <mapping resource="hibernate/MixedServiceAction.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

The first four property elements contain the necessary configuration for the JDBC connection. After, it is the configuration of the C3P0 connection pooling tool. The dialect property element specifies the particular SQL variant Hibernate generates. Hibernate's automatic session management for persistence contexts is configured to use the Hibernate built in SessionFactory and the thread execution to track the current session. The `hbm2ddl.auto` option turns on automatic updating of database schemas. Finally, we add the mapping files for persistent classes to the configuration – these are Mixed`ServiceOrder.hmb.xml` and Mixed`ServiceAction.hbm.xml`, at $SOSA_CONF/`hibernate` directory.

Hibernate needs to know how to load and store objects of the persistent class. This is where the Hibernate mapping file comes into play. The mapping file tells Hibernate what table in the database it has to access, and what columns in that table it should use.

SOSA needs to know the configuration of persistence by means of its configuration file at $SOSA_CONF/`sosa_conf.xml`. Below is a configuration example:

```
<Persistence type="persistence" name="PM"
className="com.hp.sosa.modules.sosamodule.persistence.mixedfilehibernate.MixedFileHibernateSosaPersister
                <Parameter name="persistence.hibernate.config.file"
value="conf/hbm_mixedpersistence.cfg.xml"/>
                <Parameter name="persistence.max.cache.size" value="2000000"/>
                <Parameter name="persistence.file.cache.size" value="20000000"/>
                <Parameter name="persistence.mixed.file.hibernate.max.time.file.mode" value="600000"/>
                <Parameter name="persistence.file.number.harddisk" value="4"/>
                <Parameter name="persistence.file.base.path0" value="filepersistence"/>
                <Parameter name="persistence.file.base.path1" value="/diska/persistence"/>
                <Parameter name="persistence.file.base.path2" value="/diskb/persistence"/>
                <Parameter name="persistence.file.base.path3" value="/diskc/persistence"/>
        </Persistence>
```

The first two parameters contain the configuration for the cache:

`persistence.max.cache.size`: the maximum number of elements that can be kept in cache. This parameter should be high enough to keep a big number of elements in cache but not too high to keep an efficient performance, so it depends on the machine where SOSA is installed. Values higher than 1500 are not recommended.

`persistence.time.max.in.cache`: and the maximum time (expressed in milliseconds) that each element can remain in cache before being removed.

`persistence.mixed.file.hibernate.max.time.file.mode`: indicates the maximum time that an order can stay saved as file before to change to hibernate.

`persistence.file.number.harddisk:` number of hard disk to be used

`persistence.file.base.path0:` path name of position 0. In case hard disk is configured more than one, it's also possible to configure path1,path2, …

The following parameter:

`persistence.hibernate.config.file`: indicates the Hibernate configuration file that should be loaded to initialize Hibernate.

In this persistence is configured into SOSA then the file $JBOSS_HOME/server/diagnostic/deploy/hpovact.sar/activator.war/properties/ hbm_persistence.properties needs to have next content

```
persistence.hibernate.config.file =
/etc/opt/OV/ServiceActivator/config/sosa/hbm_mixedpersistence.cfg.xml
```

8 Managers

Managers in SOSA are defined to listen to the status changes of the Service Sosa objects and do a particular process when they occur. Developers can extend the abstract class `Manager` to create there own managers and make them listen to the desired status change events. Each created manager needs to be defined in the SOSA configuration file with a `<Manager>` tag inside the `<Managers>` tag.

8.1 History Manager

This is the manager used to listen to the change status event of the SOSA services. Particularly, it listens to the status changes to RETURNED of the root Service Orders – a root Service Order is one that has no parent –, to pass them and the entire tree service to the History register. It has been developed using the Hibernate tool in a database-based register.

Below is an example of the History Manager configuration including all its configurable parameters:

```
<Manager name="HISTORY"
className="com.hp.sosa.modules.sosamodule.managers.history.HibernateHistoryManager">
  <Parameter name="history.hibernate.config.file" value="hbm_history.cfg.xml"/>
  <Parameter name="history.hibernate.force.history" value="false"/>
  <Parameter name="history.hibernate.force.history.dont.wait" value="true"/>
  <Parameter name="history.hibernate.max.store.interval" value="3000"/>
  <Parameter name="history.hibernate.enqueue.wait.interval" value="100"/>
  <Parameter name="history.hibernate.num.enqueued.so.to.reject" value="200"/>
  <Parameter name="history.hibernate.num.enqueued.so.to.notify" value="100"/>
  <Parameter name="persistence.file.number.harddisk" value="4"/>
  <Parameter name="history.hibernate.file.number.harddisk" value="4"/>
  <Parameter name="history.hibernate.file.base.path0" value="filehistory"/>
  <Parameter name="history.hibernate.file.base.path1" value="/diska/history"/>
  <Parameter name="history.hibernate.file.base.path2" value="/diskb/history"/>
  <Parameter name="history.hibernate.file.base.path3" value="/diskc/history"/>
</Manager>
```

Inside the `Manager` tags, it is required to indicate the name of the manager and the class name of its implementation, using the `name` and `className` attributes. These are the configurable parameters in the Hibernate History Manager:

`history.hibernate.config.file`: indicates the Hibernate configuration file for this manager's Hibernate Session Factory.

`history.hibernate.force.history`: when it set to true, Service Order are always passed to history, never rejected. If the queue is full, the process waits the `history.hibernate.max.store.interval` time to try to queue the Service Order again.

`history.hibernate.max.store.interval`: it is the maximum time (expressed in milliseconds) to wait between checks of the pending queue.

`history.hibernate.enqueue.wait.interval`: this means the amount of time in milliseconds to wait to retry to queue a Service Order after force a retry of the queue checker thread.

`history.hibernate.num.enqueued.so.to.reject`: when the queue of pending elements to pass to History reaches the size indicated by this parameter, the new requests are rejected unless the `history.hibernate.force.history` parameter is set to true, in that case the manager forces a new iteration of the thread and waits until the queue is drained.

`history.hibernate.num.enqueued.so.to.notify`: when the queue of pending elements to pass to History reaches the size indicated by this parameter, the manager forces a new iteration of the thread to drain the queue.

`history.hibernate.file.number.harddisk:` number of hard disk to be used

`history.hibernate.file.base.path0:` path name of position 0. In case hard disk is configured more than one, it's also possible to configure path1,path2, …

8.1.1 Partition history tables

In case the partition table needs to be partitioned the script "history_partition_creation.sql" has to be executed. NOTE: all data in history will be lost.

After that the parameter <property name="hibernate.hbm2ddl.auto">***update***</property> has to contain the value "update" in hbm_history.cfg.xml file.

In the history manager configuration next parameters have to be configured:

`history.hibernate.has.partitions:` value true to start partition control system. This means, the history manager will manage creation and deletion of partitions

`history.hibernate.partitions.in.advance: number of partition created in advance. Default value 7.`

`history.hibernate.partitions.max.days: number of days that the history will keep the history. Default 30.`

8.1.2 Create custom columns

The tables HISTORY_HBM_SO and HISTORY_HBM_SA can be customized adding new columns base on input, output, rollback or context maps. To add a new column the files HistoryServiceAction.hbm.xml or HistoryServiceOrder.hbm.xml have to be configured.

Next line has to be added to create a new column:

<property name="**parametername**" type="string" column="**COLUMN_NAME**" access="com.hp.sosa.modules.sosamodule.managers.history.HistoryPropertyAccessor"/>

In this example, the parameter will be find in all maps in the next order; output, input, rollback and context. Also there's the value is expected to be String and it will be saved as VARCHAR in database.

In case we want to do a type conversion, always from String to Date, Int, Long or Boolean next configuration has to be defined.

<property name="**boolean__**parametername" type="boolean" column="**COLUMN_NAME**"
access="com.hp.sosa.modules.sosamodule.managers.history.HistoryPropertyAccessor"/>

<property name="**int__**parametername" type="int" column="**COLUMN_NAME**"
access="com.hp.sosa.modules.sosamodule.managers.history.HistoryPropertyAccessor"/>

<property name="**long__**parametername" type="long" column="**COLUMN_NAME**"
access="com.hp.sosa.modules.sosamodule.managers.history.HistoryPropertyAccessor"/>

<property name="**date__**parametername" type="timestamp" column="**COLUMN_NAME**"
access="com.hp.sosa.modules.sosamodule.managers.history.HistoryPropertyAccessor"/>

The Boolean string value has to be true/false or yes/no.

The date format has to be yyyy/MM/dd HH:mm:ss , it's possible to customize this format setting the parameter history.hibernate.default.date.format.new.columns in sosa-module.properties using the SimpleDateFormat format.

Also, it's also possible to force use only one Map. In that case, next are the configuration examples:

<property name="**input__**parametername" type="string" column="**COLUMN_NAME**"
access="com.hp.sosa.modules.sosamodule.managers.history.HistoryPropertyAccessor"/>

<property name="**output__**parametername" type="string" column="**COLUMN_NAME**"
access="com.hp.sosa.modules.sosamodule.managers.history.HistoryPropertyAccessor"/>

<property name="**context__**parametername" type="string" column="**COLUMN_NAME**"
access="com.hp.sosa.modules.sosamodule.managers.history.HistoryPropertyAccessor"/>

<property name="**rollback__**parametername" type="string" column="**COLUMN_NAME**"
access="com.hp.sosa.modules.sosamodule.managers.history.HistoryPropertyAccessor"/>

Finally, maps and format can be combined, first the map has to be defined and after that the format. For example,

<property name="**input__boolean__**parametername" type="**boolean**"
column="**COLUMN_NAME**"
access="com.hp.sosa.modules.sosamodule.managers.history.HistoryPropertyAccessor"/>

In order to show one of these parameters in the Solution Container search view filters, a block like the following must be added to the `sosa3.properties` file in `$JBOSS_HOME/server/diagnostic/deploy/hpovact.sar/activator.war/properties`:

For Service Orders:

```
sosa.history.serviceorder.field0.name=FOO
sosa.history.serviceorder.field0.attr=input__foo
sosa.history.serviceorder.field0.type=STRING

sosa.history.serviceorder.field1.name=BAR
sosa.history.serviceorder.field1.attr=output__bar
sosa.history.serviceorder.field1.type=INTEGER
```

For service actions:

```
sosa.history.serviceaction.field0.name=BAZ
sosa.history.serviceaction.field0.attr=context__baz
sosa.history.serviceaction.field0.type=BOOLEAN

sosa.history.serviceaction.field1.name=QUX
sosa.history.serviceaction.field1.attr=rollback__qux
sosa.history.serviceaction.field1.type=LONG
```

If custom parameters are defined in `sosa3.properties`, displaying of the default ones can be omitted. In order to hide default parameters the following properties can be used:

```
sosa.history.serviceorder.showdefaultfields=false
```

for service orders and

```
sosa.history.serviceaction.showdefaultfields=false
```

for service actions.

8.2 Performance Manager

The performance manger can be added to monitor the system and control how many change of status has been in the interval period. With the relation between time and number of change status we can view in which status we spent more time. This module is only for debugging purpose and then it should be avoided to use in production environment.

Below is an example of the Performance Manager configuration:

```
<Manager
className="com.hp.sosa.modules.sosamodule.managers.performance.PerformanceManager"
name="PERFORMANCE">
  <Parameter name="performance.manager.interval" value="10000"/>
  <Parameter name="performance.manager.status.timing" value="true"/>
  <Parameter name="performance.manager.active.sae.timing" value="true"/>
</Manager>
```

There are three configurable parameters in the Performance Manager which can be indicated inside `<Parameter>` tags with the `name` and `value` attributes:

`performance.manager.interval`: The time interval (in milliseconds) between performance logs.

`performance.manager.status.timing`: It indicates whether to monitor the SOSA services status changes.

`performance.manager.active.sae.timing`: It indicates whether to monitor the SOSA services persistence operations: select, insert, update and delete.

Also, it is necessary to configure log4j file properly. The configuration file is at `$SOSA_HOME/properties/sosa-core-log4j.properties`. It is necessary to add next lines:

```
### direct messages to file performance.log ###
log4j.appender.performance=org.apache.log4j.RollingFileAppender
log4j.appender.performance.File=log/performance.log
log4j.appender.performance.MaxFileSize=20MB
log4j.appender.performance.MaxBackupIndex=20
log4j.appender.performance.layout=org.apache.log4j.PatternLayout
log4j.appender.performance.layout.ConversionPattern=%d{ABSOLUTE} [%t] %5p %c{1}:%L - %m%n

log4j.logger.com.hp.sosa.modules.sosamodule.managers.performance.PerformanceManager=INFO,
performance
```

8.3 Performance Status Manager

The performance status manager saves the last 24 hours performance statistics. It publishes a rmi interface to access the last minutes status for:

- Protocol adapters
- ServiceActionExecutors
- Queues
- ServiceAction
- ServiceOrder

Below is an example of the Performance Manager configuration:

```
<Manager
className="com.hp.sosa.modules.sosamodule.managers.performance.PerformanceStatusManager"
name="PERFORMANCE_STATUS">
 <Parameter name="performance.manager.interval" value="60000"/>
 <Parameter name="performance.manager.service.order.only.root" value="false"/>
</Manager>
```

There are three configurable parameters in the Performance Manager which can be indicated inside `<Parameter>` tags with the `name` and `value` attributes:

`performance.manager.interval`: The time interval (in milliseconds) between performance status file will be save in case SOSA restart to be able to recover it. logs.

`performance.manager.service.order.only.root:` It indicates if only root ServiceOrder will be created statistics.

8.4 Creating new manager

To create a new manager the class com.hp.sosa.modules.sosamodule.managers.Manager needs to be extended. The performance manger can be added to monitor the system and control how many change of status.

This is an empty example of new manager implementation

```
import com.hp.sosa.exceptions.SosaException;
import com.hp.sosa.modules.sosamodule.Constants;
import com.hp.sosa.modules.sosamodule.elements.ServiceAction;
import com.hp.sosa.modules.sosamodule.elements.ServiceOrder;
import com.hp.sosa.modules.sosamodule.managers.Manager;

public class NewExampleManager  extends Manager{

	private final String[] serviceOrderStatusListened
=
{Constants.STATUS_CREATED,Constants.STATUS_BUILDED,Constants.STATUS_SCHEDULED
		,Constants.STATUS_PROCESSED,Constants.STATUS_RETURNED,
Constants.STATUS_PAUSE,Constants.STATUS_WAIT_CHILD};

	private final String[] serviceActionStatusListened =
{Constants.STATUS_CREATED,Constants.STATUS_BUILDED,Constants.STATUS_ERROR,
		Constants.STATUS_SCHEDULED,
Constants.STATUS_PROCESSING,Constants.STATUS_PROCESSED,
Constants.STATUS_PAUSE,Constants.STATUS_ENQUEUED};

	public void changeStatus(ServiceOrder so, String currentStatus,
			String newStatus) throws SosaException {
		//make the action implies for this change of status

	}

	public void changeStatus(ServiceAction sa, String currentStatus,
			String newStatus) throws SosaException {
		//make the action implies for this change of status

	}

	public void finish() {
	}

	public String[] getServiceActionStatusListened() {
		return serviceActionStatusListened;
	}

	public String[] getServiceOrderStatusListened() {
		return serviceOrderStatusListened;
	}

	public void init(com.hp.sosa.modules.sosamodule.conf.Manager
managerConf)
			throws SosaException {
		//init configuration

	}

	public void refreshConfiguration(
			com.hp.sosa.modules.sosamodule.conf.Manager managerConf) {
		//refresh configuration
```

```
        }

}
```

public String[] getServiceActionStatusListened() : it's very important return only the status this manager wants to listen when the ServiceAction change of status. For example, the history manager returns empty because there's no need to make any action for ServiceActions.

public String[] getServiceOrderStatusListened() : it's very important return only the status this manager wants to listen when the ServiceOrder change of status. For example, the history manager returns only the state RETURNED because only make an action when the ServiceOrder change to this status.

public void init(com.hp.sosa.modules.sosamodule.conf.Manager managerConf) : when SOSA stars call this method to initialize the manager.

public void refreshConfiguration(com.hp.sosa.modules.sosamodule.conf.Manager managerConf) : in case it's possible to refresh the configuration fill this method, if not leave empty.

public void finish() : when SOSA stops call this method and the manager can uncofigure or close the connections.

public void changeStatus(ServiceOrder so, String currentStatus, String newStatus) : every time that a ServiceOrder change the status to newStatus parameter, SOSA will call this method in case the method getServiceOrderStatusListened include this new status. It's important to know this method is inside of a transaction, then all change into so will be saved after this method.

public void changeStatus(ServiceAction sa, String currentStatus, String newStatus) : every time that a ServiceAction change the status to newStatus parameter, SOSA will call this method in case the method getServiceActionStatusListened include this new status. It's important to know this method is inside of a transaction, then all change into sa will be saved after this method.

9 Executing Service Orders

When Service Orders are sent to SOSA to be processed, each service of the tree service goes through different states while being executed.

The execution of Service Orders last from the protocol adapter inserts them into SOSA to SOSA responds the protocol adapter.

9.1 Service State Diagrams

The state of a Service Order may have the following values:

Name	Description
CREATED	A new Service Order is created in SOSA using the input parameters.
BUILT	The tree service associated with the Service Order is built from service catalog.
SCHEDULED	The Service Order is scheduled waiting for being processed.
WAIT_CHILD	The Service Order is waiting for children execution.
PROCESSED	All children of Service Order have been executed.
PAUSE	The Service Order is waiting for user interaction.
RETURNED	The Service Order is processed and the client has been notified.

Next figure shows the Service Order state diagram:

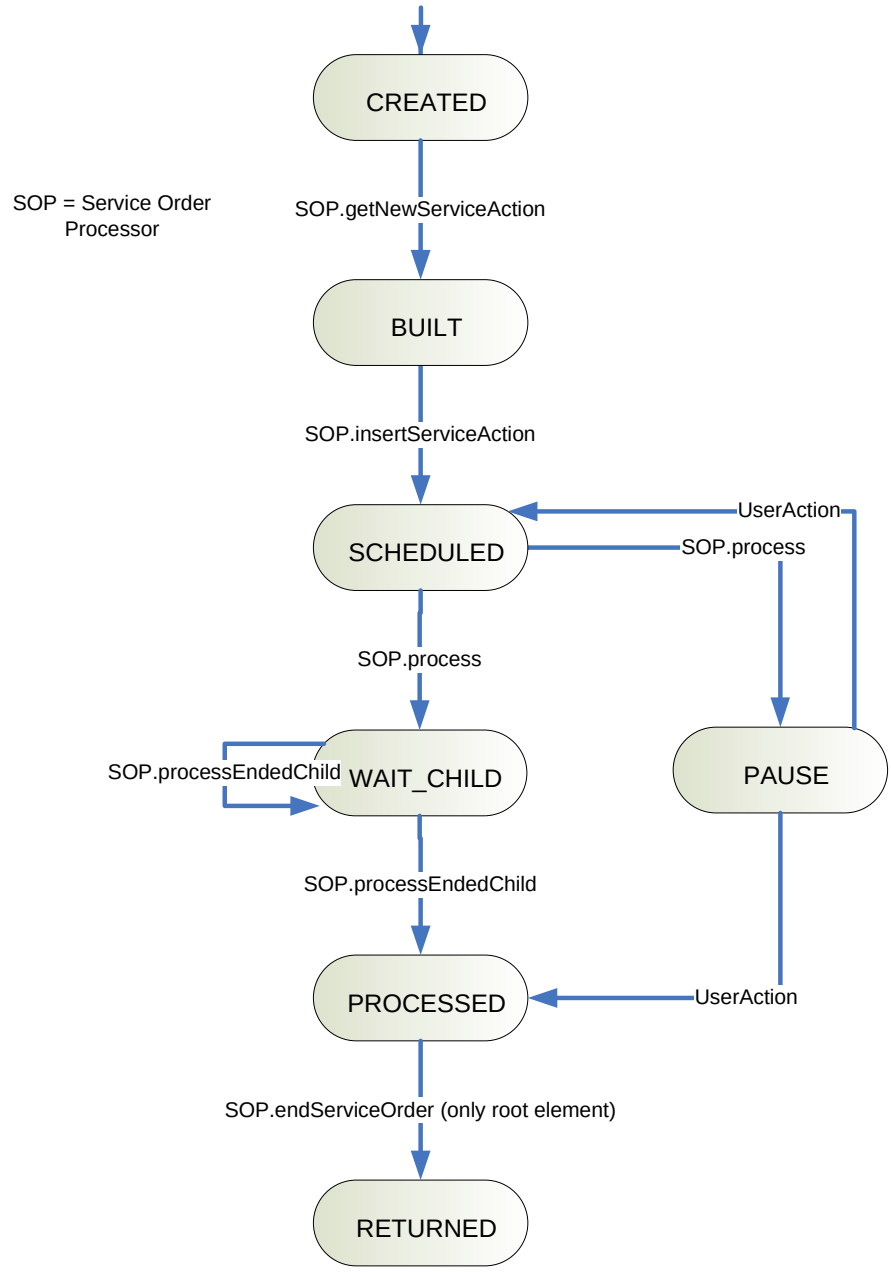

Fig. 3: Service order state diagram

The state of a Service Action may have the following values:

Name	Description
CREATED	A new Service Action is created in SOSA using the input parameters.
BUILT	The Service Action is built from service catalog.
SCHEDULED	The Service Action is scheduled waiting for being queued.

ENQUEUED	The Service Action is queued waiting for being executed.
PROCESSING	The Service Action is being processed.
PROCESSED	The Service Action has finished its execution.
PAUSE	The Service Action is waiting for user interaction.
ERROR	An error occurred while executing the Service Action.

Next figure shows the Service Action state diagram:

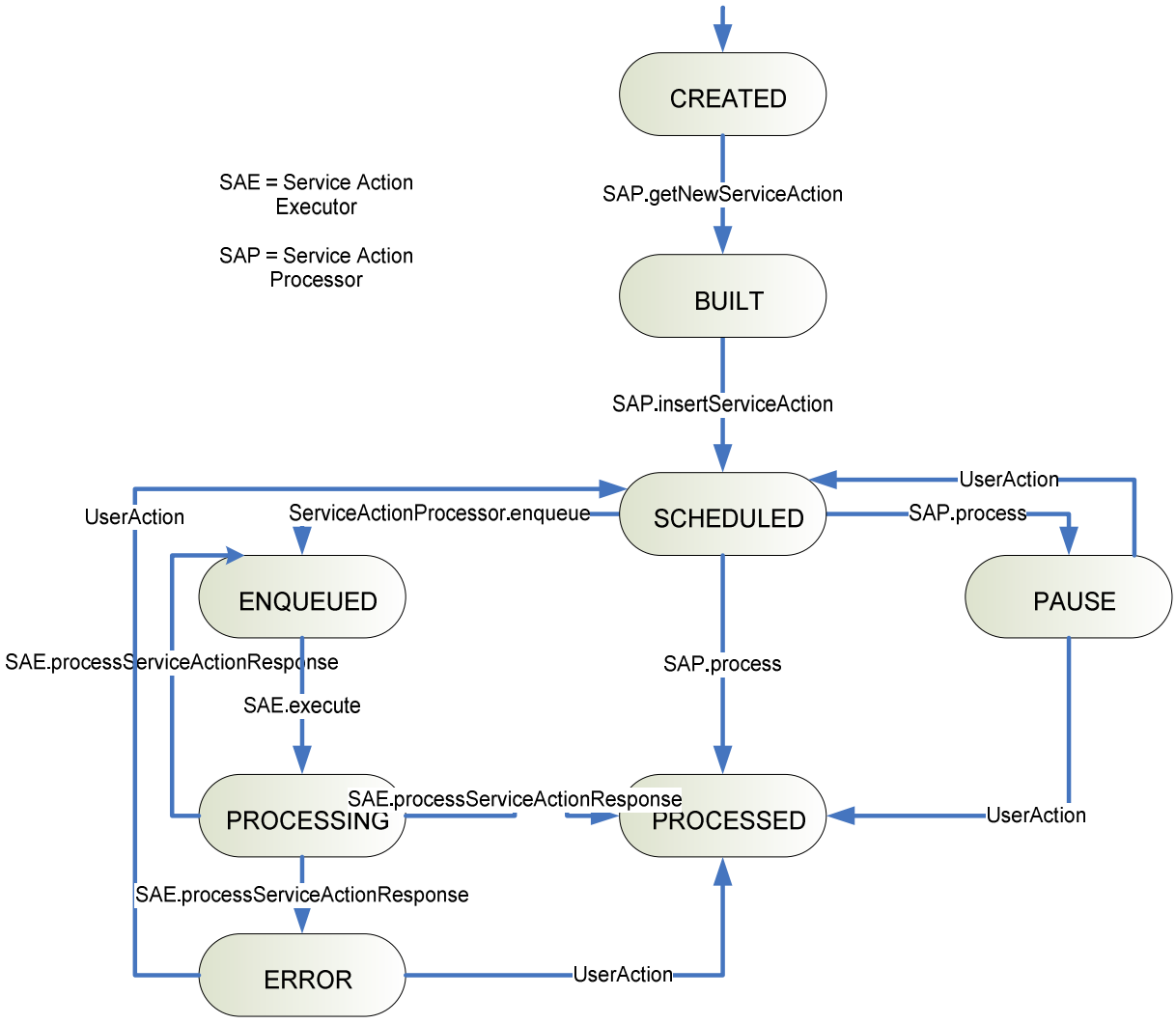

Fig. 4: Service action state diagram

10 Time Window Module

The Time Window Module provides to SOSA a way to execute actions periodically on several components. The actions in SOSA may be applied to Protocol Adapters, Listeners, Queues and Service Action Executors.

For instance a scheduled action could be, everyday lock a queue at 0:00 and unlock it at 8:00.

Also, the Time Window Module offers the possibility to define exclusion calendars when actions are not executed.

Next table shows the allowed operations that may be scheduled.

Component	Actions
Protocol Adapter	Pause, Resume
Listener	Start, Stop
Queue	Open, Close, Lock, Unlock
Service Action Executor	Lock, Unlock

All the scheduling plans are defined by a day range (specific date or expression time like "MON-WED, FRI"). Given the day range, can be assigned as many events as needed. Each event is composed by one or two simple tasks (a start and optionally a stop action). The event parameters are the following (note that all parameters are mandatory except those that optional is specified):

Type: the SOSA element that will be modified. List of values: Protocol Adapter, Queue, Listener and Service Action Executor.

Element name: name of the element to interact with.

Start event: the action to be executed on the SOSA element (start, lock, stop…).

Start time: the time in which the action will be executed (the format is HH:MM:SS, HH:MM or HH) on the start event.

Stop event (optional): the second action to be executed on the SOSA element. It is not a mandatory parameter.

Stop time (optional): if stop event is defined, this field represents the time in which the action will be executed (the format is HH:MM:SS, HH:MM or HH).

Calendar Exclusions (optional): you can define as many exclusion dates as you need and group them into a calendar. If you specify a calendar and an event that must be executed in a day that belongs to the calendar, the event action will not be executed on that exclusion date.

11 Tricks and recommendations

Here there're some recommendations or useful information to make some task.

11.1 Logging

Every time that a new class is created it's highly recommended to create his log.

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class NewClass{
      private static Log log = LogFactory.getLog(NewClass.class);
}
```

The whole SOSA log is defined into the file $SOSA_HOME/properties/sosa-4logj.propertis In this file it's possible to add more appenders, change the log level, … It's possible to change this file when SOSA is running because new changes will be loaded after a while.

11.2 Get a ServiceOrder or ServiceAction

Usually is required to load a specific ServiceOrder or ServiceAction. If there's no need to modify is recommended to load on read only. It's only necessary the id of the object.

```
ServiceOrder so = ServiceElementsManager.getServiceOrderReadOnly(ssid);

ServiceAction sa = ServiceElementsManager.getServiceActionReadOnly(ssid);
```

11.3 Modify a ServiceAction or ServiceOrder

To modify a ServiceAction or ServiceOrder a new transaction need to be open and close. For this purpose we need to load the object in "write mode" and finally write or undo the object. It's very important always to make the write or undowrite, if not the transaction will keep open. Here is an example:

```
ServiceOrder so = null;
      try{
so = (ServiceOrder) ServiceElementsManager.getServiceOrderToWrite(ssid);
if (so == null) throw new SosaException("Cannot get service '" + ssid +
"'");
//so modifications
ServiceElementsManager.writeServiceOrder(so);
so = null;
      }catch(SosaException e){
```

```
      log.fatal("cannot insert the service order '" + so +"'",e);
      }
      finally{
      if (so != null) ServiceElementsManager.undoWriteServiceOrder(so);
      }
```

11.4 Load entire tree order

In some case we need to load the entire tree order to check or to find or check an specific status from a object. This is an example of two method who save the tree into a Map.

```
private Map loadSoTree(String ssid)
      {
            Map serviceSosas = new HashMap();
            if(ServiceElementsManager.isServiceOrder(ssid))
            {
                  try
                  {
                        com.hp.sosa.modules.sosamodule.elements.ServiceOrder
so =

      ServiceElementsManager.getServiceOrderReadOnly(ssid);
                        if(so != null)
                              serviceSosas.putAll(loadSoTree(so));
                        else
                              throw new SosaException("Cannot load Service
with SSID '" + ssid + "'.");
                  }
                  catch (SosaException se)
                  {
                        log.error("Exception trying to load a Service Order
tree form persistence. Tree won't be loaded. ", se);
                        serviceSosas.clear();
                  }
            }
            return serviceSosas;
      }

      private Map
loadSoTree(com.hp.sosa.modules.sosamodule.elements.ServiceOrder so)
      {
            Map serviceSosas = new HashMap();
            if(so != null)
            {
                  try
                  {
                              serviceSosas.put(so.getSsid(),so);

                        log.debug("Service Order added to tree: " +
so.getSsid());
                        if(so.getChilds() != null)
```

```
                        {
                              Iterator childrenIt =
so.getChilds().iterator();
                              while(childrenIt.hasNext())
                              {
                                    String childSsid =
(String)childrenIt.next();

      if(ServiceElementsManager.isServiceAction(childSsid))
                                    {

      com.hp.sosa.modules.sosamodule.elements.ServiceAction sa =

      ServiceElementsManager.getServiceActionReadOnly(childSsid);
                                          if(sa != null)
                                          {

                              serviceSosas.put(sa.getSsid(),sa);

                                                log.debug("Service Action
added to tree: " + sa.getSsid());
                                          }
                                    }
                                    else
if(ServiceElementsManager.isServiceOrder(childSsid))

      serviceSosas.putAll(loadSoTree(childSsid));
                              }
                        }
                  }
                  catch (SosaException se)
                  {
                        log.error("Exception trying to load a Service Order
tree form persistence.", se);
                        serviceSosas.clear();
                  }
            }
            return serviceSosas;
      }
```

11.5 Implementing a ServiceActionExcutor Selector

The state of a Service Order may have the following

11.6 Creating new database connection

To create a new database pool is recommended to use the default SOSA utility DataBasePoolManager. First it's required to create the pool and after that there're two methods to get and return the connections.

Here is an example of creating a new pool.

```
                    DataBasePoolManager.createNewDataBasePool(poolName, user,
password, host, port, instance);
                    DataBasePoolManager.setJdbcDriver(poolName, jdbcDriver);
                    DataBasePoolManager.setDriverName(poolName, driverName);
                    DataBasePoolManager.setInitialSize(poolName, initialSize);
                    DataBasePoolManager.setMaxActive(poolName, maxActive);
                    DataBasePoolManager.setMaxIdle(poolName, maxIdle);
                    DataBasePoolManager.setMinIdle(poolName, minIdle);
                    DataBasePoolManager.setMaxWait(poolName, maxWait);
                    DataBasePoolManager.startDataBasePool(poolName);
```

To get and return a connection there're next two methods:

```
con = DataBasePoolManager.getConnection(poolName);

DataBasePoolManager.releaseConnection(poolName, con);
```

It's important to release the connection once the work is finished.

11.7 Avalanche Control

An optional mechanism of avalanches control has been included to avoid massive insertion of orders that could make SOSA work inefficiently. This control is configured at file $SOSA_HOME/`properties/sosa-module.properties`, like the following example:

```
service.element.manager.max.insert.serviceorder.persistable=80
service.element.manager.max.insert.serviceorder.nonpersistable=1500
service.element.manager.max.insert.serviceorder.time=5000
service.element.manager.max.insert.serviceaction.persistable=80
service.element.manager.max.insert.serviceaction.nonpersistable=1500
service.element.manager.max.insert.serviceaction.time=5000
```

The meaning of the parameters is as follows:

`service.element.manager.max.insert.serviceorder.time`: The period of time in milliseconds before the count of inserts is reset to zero.

`service.element.manager.max.insert.serviceorder.persistable`: The maximum number of persistent Service Orders that can be inserted into SOSA every period of time defined before.

`service.element.manager.max.insert.serviceorder.nonpersistable`: The maximum number of non-persistent Service Orders that can be inserted into SOSA every period of time defined before.

`service.element.manager.max.insert.serviceaction.time`: The period of time in milliseconds before the count of inserts is reset to zero.

`service.element.manager.max.insert.serviceaction.persistable`: The maximum number of persistent Service Actions that can be inserted into SOSA every period of time defined before.

`service.element.manager.max.insert.serviceaction.nonpersistable`: The maximum number of non-persistent Service Actions that can be inserted into SOSA every period of time defined before.

When an external system tries to insert a Service and it exceeds the limit configured, SOSA delays its response the necessary milliseconds until the next period of control time begins.