HPSA Extension Pack

SOSA

Release V6.1

Legal Notices

Document id: EP-pd001603

Table of Contents

In This Guide

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

Audience

The audience for this guide is the developers of EP applications and solutions. The EP 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

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/WFM.xml`
	Process names	Check to see if `WFM` 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\`
`$JBOSS_HOME`	HOME The install location for JBoss. The UNIX location is `/opt/HP/jboss` The Windows location is `<drive>:\HP\jboss`
`$JBOSS_DEPLOY`	The install location of the Service Activator J2EE components. The UNIX location is `/opt/HP/jboss/stanalone/deployments` The Windows location is `<drive>:\HP\jboss\stanalone\deployments`
`$ACTIVATOR_DB_USER`	The database user name you define. Suggestion: `ovactivator`
`$ACTIVATOR_SSH_USER`	The Secure Shell user name you define. Suggestion: `ovactusr`
`$SOSA_HOME`	The install base location of SOSA. The default UNIX location is `/opt/OV/ServiceActivator/EP/SOSA` The default Windows location is <drive>:\HP\OpenView\ServiceActivator\EP\SOSA\
`$SOSA_BIN`	The install location of specific SOSA binary files. The default UNIX location is `/opt/OV/ServiceActivator/EP/SOSA/bin` The default Windows location is `<drive>:\HP\OpenView\ServiceActivator\EP\SOSA\bin\`
`$SOSA_CONFIG`	The install location of specific SOSA configuration files. The default UNIX location is `/opt/OV/ServiceActivator/EP/SOSA/conf` The default Windows location is <drive>:\HP\OpenView\ServiceActivator\EP\SOSA\conf\
`$ECP_HOME`	The install base location of Equipment Connections Pool. The default UNIX location is `/opt/OV/ServiceActivator/EP/ECP` The default Windows location is <drive>:\HP\OpenView\ServiceActivator\EP\ECP\
`$ECP_BIN`	The install location of specific Equipment Connections Pool binary files. The default UNIX location is `/opt/OV/ServiceActivator/EP/ECP/bin` The default Windows location is <drive>:\HP\OpenView\ServiceActivator\EP\ECP\bin\
`$ECP_ETC`	The install location of specific Equipment Connections Pool configuration files. The default UNIX location is

	`/opt/OV/ServiceActivator/EP/ECP/conf` The default Windows location is <drive>:\HP\OpenView\ServiceActivator\EP\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
WFM: 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

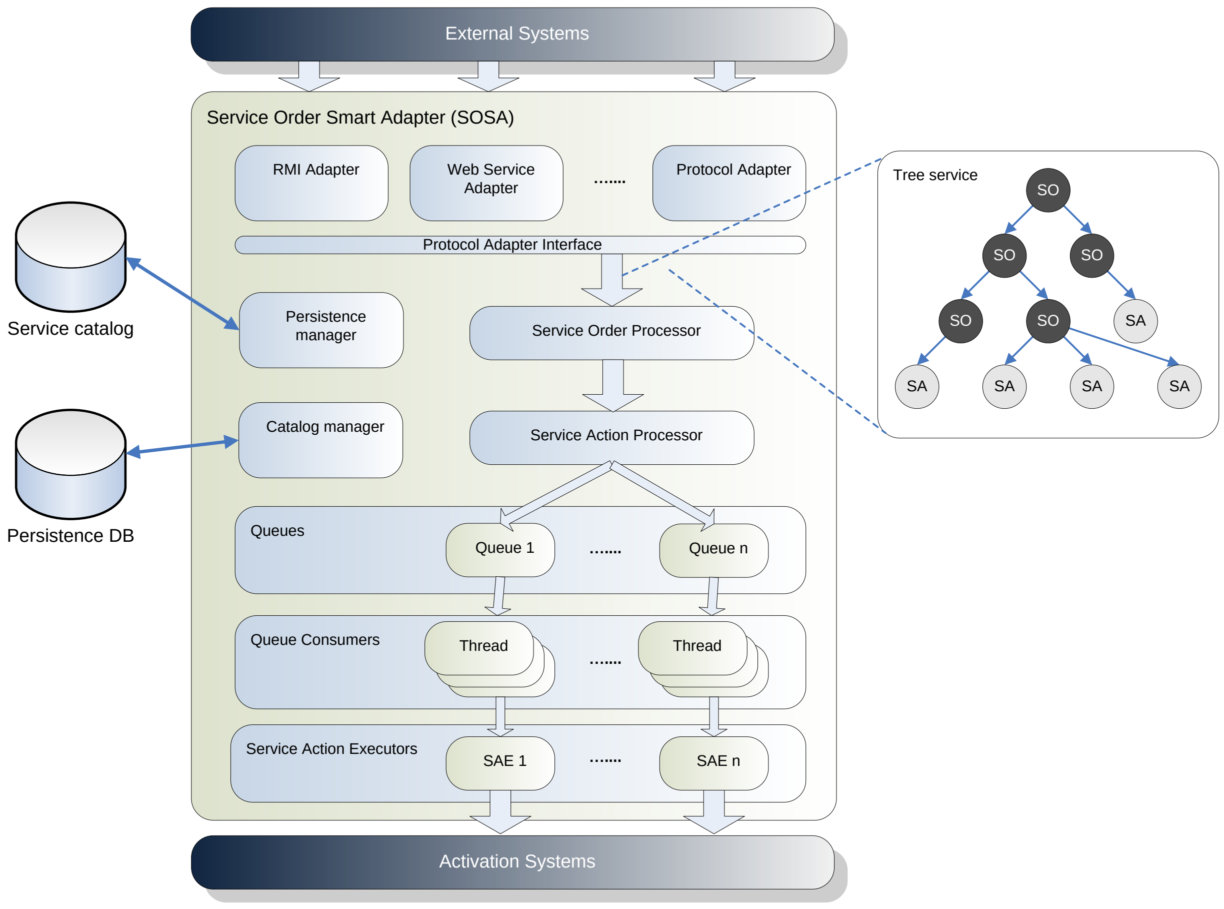
2.1 Functionality

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.

2.2 Architecture

The figure below 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 WFM 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.


External Systems
Service Order Smart Adapter (SOSA)
RMI Adapter
Web Service Adapter
....
Protocol Adapter
Protocol Adapter Interface
Service catalog
Persistence manager
Service Order Processor
Persistence DB
Catalog manager
Service Action Processor
Queues
Queue 1
....
Queue n
Queue Consumers
Thread
....
Thread
Service Action Executors
SAE 1
....
SAE n
Activation Systems
Tree service
SO
SO
SO
SO
SO
SA
SA
SA
SA
SA

3 Modules

SOSA is based on modules. This, apart from providing scalability, sets a clear separation among the different features provided by this tool. All the modules that are part of SOSA are configured in `$SOSA_CONFIG/sosa.xml`. The next sections describe the currently existing modules and how must they be configured.

3.1 Main Module

This module is called `sosaModule` (this name is required and cannot be modified) and it is responsible of initializing the different SOSA components (such as managers, protocol adapters, service catalog, and the persistence layer) when SOSA is started up. In the same line, it is also responsible for stopping them when SOSA is shutting down.
This module accepts the next parameters:

- `sosa.instance.name`: The name of the SOSA instance. It is only required when SOSA is configured to work in high availability.
- `sosa.conf.file`: Indicates the path of SOSA configuration file. It is an optional parameter and if not defined it defaults to `$SOSA_CONFIG/sosa_conf.xml`.
- `jetty.start`: Indicates whether the web service provided with SOSA must be started up or not. It accepts the values `true` to start up the web service and `false` to skip it. It is an optional parameter and if not defined defaults to `false`.
- `sosa.recover.service.order.threads`: The number of threads that will be started up to restore the persisted services once SOSA has been started up. These threads will stop working when there are no more persisted services to restore. It is an optional parameter and defaults to 20.
- `sosa.action.saver.enable`: Indicates whether SOSA has to save the state of each component (Managers, Protocol Adapters, Queues and Service Action Executors) into a XML formatted file (see parameter `sosa.action.saver.file`). It accepts the values `true` (SOSA will write the XML file) and `false`. It is an optional parameter and defaults to `true`.
- `sosa.action.saver.file`: A XML formatted file where SOSA will write the state its Managers, Protocol Adapters, Queues and Service Action Executors. This file is managed by SOSA and cannot be modified from outside. It is primarily intended to provide that information quickly and easily. This is an optional parameter and if not specified defaults to `$SOSA_CONFIG/sosa_action_saver.xml`.
- `sosa.client.service`: The name of the RMI service published by SOSA to manage services externally. It is an optional parameter and its default value is `SosaClient`.

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

3.2 Time Window Module

The Time Window Module offers a way to schedule tasks or actions to be applied to SOSA components. Further sections related to the behavior of the Time Window Module will provide a detailed description of its features.

This module is called `timeWindowModule` (this name is mandatory and cannot be modified) and requires the next initial parameters to configure the database pool which access the database where the scheduling tasks will be stored:

- `db.pool.name`: The database pool name. It is a mandatory parameter.
- `db.user`: The user name. It is a mandatory parameter.
- `db.password`: The user's password. It must be encrypted, never in clear text. Service Activator provides an encrypt utility to generate encrypted passwords. It is a mandatory parameter.
- `db.jdbc.driver`: The Java class of the database driver to use. Drivers are different for Oracle, EnterpriseDB and PostgreSQL. It is a mandatory parameter.
- `db.url`: The database URL. It is a mandatory parameter.
- `db.initialsize`: The number of connections to keep open and available since the beginning.
- `db.maxactive`: The cap on the total number of active instances from my pool. Use a negative value for an infinite number of instances.
- `db.maxidle`: The cap on the number of *idle* instances in the pool. Use a negative value to indicate an unlimited number of idle instances.
- `db.minidle`: The minimum number of objects allowed in the pool before the evictor thread (if active) spawns new objects.

Note: No objects are created when: `numActive + numIdle >= maxActive`.

- `db.maxwait`: The maximum amount of time (in millis) an object should be blocked before throwing an exception when the pool is exhausted. When less than 0, objects may be blocked indefinitely.

```
<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="myuser" />
    <Parameter name="db.password" value="mypassword" />
    <Parameter name="db.jdbc.driver" value="oracle.jdbc.driver.OracleDriver" />
    <Parameter name="db.url" value="jdbc:oracle:thin:@//myhost:1521/orcl" />
    <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>
```

3.2.1 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`

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

- `name`: The name of the Service Action Executor.
- `className`: The complete class name where the Service Action Executor is implemented.
- `max_parallelism`: The maximum number of concurrent executions that can be handled by the Service Action Executor.

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

4.1 WFM Service Action Executors

In many of SOSA scenarios, Service Actions represents HPSA workflows. The WFM Service Action Executor is a SAE implementation provided by SOSA, which is able to manage the workflow execution on a HPSA WFM instance.
The communication between the WFM SAE and the WFM instance is performed via RMI. When WFM SAE starts a workflow use the RMI interface of the WFM. If the execution is made in asynchronous mode, the WFM will have to notify WFM SAE at the end of execution. Therefore, asynchronous mode requires that the WFM knows how to communicate with SAE. Later we will see how to do this.
To add a WFM SAE to SOSA a new entry in the SOSA configuration file must be added. Below is shown a configuration example of a WFM SAE.

```
<ServiceActionExecutors>
  <ServiceActionExecutor
        className="com.hp.sosa.modules.sosamodule.executors.mwfm.WFMServiceActionExecutor"
        max_parallelism="0" name="WFM_SA_EXECUTOR">
    <Parameter name="host" value="127.0.0.1"/>
    <Parameter name="port" value="2000"/>
    <Parameter name="user" value="mysystemuser"/>
    <Parameter name="password" value="mysystempassword"/>
    <Parameter name="async_interval" value="10000"/>
    <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 WFM SAE configuration parameters are the following:

- `host`: The hostname where the WFM is running.
- `port`: The port where the RMI instance of WFM is published.
- `user`: A valid user name to log on in the WFM.
- `password`: The encrypted password of the given user to log on in the WFM.
- `launch_retries`: The number of times the SAE will retry if a Service Action cannot be executed for any reason.
- `async_interval`: The time to wait (in milliseconds) until the running jobs information is retrieved from the WFM.
- `max_time_sync_job`: The maximum time in milliseconds that a work flow which has been started up in synchronous mode can be working until it is interrupted.
- `max_time_sync_job_interval`: The time to wait (in milliseconds) until the information about a work flow that has been started up in synchronous mode is retrieved from the WFM. This is required in order to check whether the work flow has spent its working time (see `max_time_sync_job`).
- `timeout`: The maximum amount of processing time for a Service Action to consider it timed out.
- `timeout_interval`: The minimum time in milliseconds between timeout checks
- `copy_cp_to_output`: Accepts the values `true` and `false` and indicates whether to copy all workflow case-packet variables into the `output_params` map when the execution finishes (`true`) or not (`false`).
- `add_extra_info`: In case the Service Action's parameter called `extra_info` has a valid value, add that value to the input parameters. The parameter `extra_info` must be plain text containing a list of name/value pairs separated by the characters configured in `extra_info_attrib_limiter` and it will be parsed and transformed into a Java Map. E.g. the text
  `variable1=value1|;|variable2=value2`
  will be transformed into the map
  `{variable1=value1,variable2=value2}`
- `extra_info_value_limiter`: The character used to assign a name to its value. Its default value is =.
- `extra_info_attrib_limiter`: The character used to separate name/value pairs. Its default value is `|;|`.

4.2 SOSA Asynchronous Response Module

When the WFM SAE is executing Service Actions in asynchronous mode, the workflow needs a way to notify SOSA from HPSA WFM when it is finished. SOSA provides this tool as a HPSA WFM module called SOSA Asynchronous Response Module.
First of all, the module has to be configured in the WFM 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"/>
</Module>
```

Where:

- `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.
- `rmi_port`: The port where publish a RMI service. Defaults to 2000.

The steps of every SOSA Asynchronous Response Module operation are:

i. First, the module tries to notify SOSA through RMI connection.
ii. If RMI connection fails, the module stores the response in persistent file to retry the response later.

4.3 Developing workflows compatible with Sosa

Once we have seen how SOSA invokes HPSA workflows using the WFM SAE, and how the WFM 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.

4.3.1 Invoking the SOSA Asynchronous Response Module from a workflow

SOSA provides two ways to invoke the response module, a WFM node and a WFM end-handler.
The recommended way is to configure the `SosaEndHandler`, because End Handlers are executed even if the workflow fails.

```
<End-Handler>
  <Name>SosaEndHandler</Name>
  <Class-Name>com.hp.ov.activator.mwfm.ep.component.builtin.SosaEndHandler</Class-Name>
</End-Handler>
```

The other and not recommended is using a WFM node `EndAsyncJob`. This node can be used like any other WFM 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 WFM message queue when the RMI service fails.

```
</Process-Node>
  <Process-Node>
  <Name>EndAsyncJob</Name>
  <Action>
    <Class-Name>com.hp.ov.activator.mwfm.ep.component.builtin.EndAsyncJob</Class-Name>
  </Action>
</Process-Node>
```

4.4 Case-packet variables in workflows related to SOSA

SOSA uses the following Case Packet 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 that 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.

4.5 Developing a Service Action Executor

Sometimes it is needed to develop a new `ServiceActionExecutor` because SOSA has to start actions in a different external system than WFM. 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 class com.hp.sosa.modules.sosamodule.executors.ServiceActionExecutor.
E.g.:

```
  public class NewExampleServiceActionExecutor extends ServiceActionExecutor
```

After that, the next methods must be implemented:

- `public boolean check()`: This method returns `true` if the SAE has connectivity and is able to start orders correctly into the external system. Otherwise the connectivity is down or the external system is down and this method will 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 up and detects that a Service Action was working in asynchronous mode on this SAE instance, SOSA calls this method to know if it can recover or not the Service Action in processing state. It will return `true` if this it is possible to keep this particular Service Action on processing state because it is assured that SOSA will receive the answer of this order. It will return `false` in the rest of the cases.
- `public boolean haveToWaitSynchronousResponse(String ssid)`: When SOSA starts up and detects that a Service Action was working in synchronous mode on this SAE instance, SOSA calls this method to know if it can recover or not the Service Action in processing state. It will return `true` if this it is possible to keep this particular Service Action on processing state because it is assured that SOSA will receive the answer of this order. It will return `false` in the rest of the cases.
- `public void init(Map initParmeters)`: This method will be called as part of the initialization of SOSA, i.e. when SOSA starts up and before any Service Action is sent to the SAE, granting this way that it will be possible to configure the Service Action. The initial parameters map contains all the parameters configured in the file `$SOSA_CONF/sosa_conf.xml` file for this SAE.

- `public void refreshConfiguration(Map initparms)`: This method will be called by SOSA when the configuration needs to be refreshed. If a SAE does not accept reconfiguration this method must be left empty. The initial parameters map contains all the parameters configured in the file `sosa_conf.xml` file for this SAE.
- `public void killService(String ssid)`: If the Service Action has been configured with a maximum time to work, this method will be called when the Service Action execution time reaches that time, so it is timed out and the SAE has to kill the Service Action in the external system if it is possible.
- `public ServiceActionResponse process(String ssid)`: This method is invoked when the Service Action is working in synchronous mode and it is responsible of executing the Service Action in the external system and returning the result via SAE.
- `public ServiceActionResponse[] process(String[] ssids)`: This method is responsible of executing Service Action Groups in synchronous mode. Note that this method in only required when the Service Action can be handled by Queues which support Groups (see the section related to Queues for more information). If the external system doesn't support execution of Service Action Groups then it is recommended to do just the lines below as part of this method:

```
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 Service Action has to be executed in synchronous mode, this meaning that the external system has to connect to SOSA to return the result. Hence, this method is just responsible of starting the Service Action in the external system. By default SOSA provides a RMI service for this purpose. The parameter called `rmiUrl` contains the location of the RMI service to which the external system has to connect.

Note: The RMI service will be an implementation of the interface
`com.hp.sosa.modules.sosamodule.executors.ServiceActionExecutorResponser`
And it is responsible of receiving the response from the external system and sending it to the SAE which handles the Service Action execution.
If the external system does not support RMI protocol then the SAE can implement any other supported protocol. The only requirement is to call the method `returnServiceActionResponse` of the SAE instance. E.g.:

```
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 does not exist for the given ssid",
            new SosaException());
   }
}
```

- `public void processAsync(String[] ssids, String rmiUrl)`: This method is responsible of executing Service Action Groups in asynchronous mode. Note that this method in only required when the Service Action can be handled by Queues which support Groups (see the section related to Queues for more information). If the external system doesn't support execution of Service Action Groups then it is recommended to do just the lines below as part of this method:

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

- `public void returnServiceActionResponse(ServiceActionResponse sar)`: This method is invoked when the external system returns the result of a Service Action which is working in asynchronous mode. As part of the internal logic inside this method an invocation to `processServiceActionResponse` has to be made in order to notify SOSA of the result of the Service Action.

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

- `public void returnServiceActionResponse(ServiceActionResponse[] sars)`: This method returns the responses of the different Service Actions handled by this SAE. The recommended implementation can be found below.

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

5 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 tag `<Queue>` 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 their 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.

5.1 Configuration

The configuration parameters described in this section are general to all the Queues defined in the system and hence they will apply to all of them.
These common configuration parameters must be set in `$SOSA_HOME/properties/sosa-module.properties`.

- `queue.class.directory`: The directory in the file system where the Java classes implementing the Queues will be stored. It is an optional parameter and if not defined defaults to `$SOSA_HOME/lib/queue`.
- `queue.serviceactionexecutor.selector`: The Java class containing the business logic to select the best Service Action Executor at a given time. It is an optional parameter and if not defined default to `com.hp.sosa.modules.sosamodule.queues.ServiceActionExecutorSelector`.
- `queue.lock.subqueue.on.lock.queue`: Accepts the values `true` and `false` and indicates whether the Sub-Queues have to be locked when their Queue is locked (`true`) or not (`false`). It is an optional parameter and if not defined defaults to `false`.
- `queue.unlock.subqueue.on.unlock.queue`: Accepts the values `true` and `false` and indicates whether the Sub-Queues have to be unlocked when their Queue is unlocked (`true`) or not (`false`). It is an optional parameter and if not defined defaults to `false`.
- `queue.open.subqueue.on.open.queue`: Accepts the values `true` and `false` and indicates whether the Sub-Queues have to be opened when their Queue is opened (`true`) or not (`false`). It is an optional parameter and if not defined defaults to `false`.
- `queue.close.subqueue.on.close.queue`: Accepts the values `true` and `false` and indicates whether the Sub-Queues have to be closed when their Queue is closed (`true`) or not (`false`). It is an optional parameter and if not defined defaults to `false`.

- `queue.group`: Accepts the values `true` and `false` and indicates whether the Queues have to be treated as Group Queues, so if `true` they will process groups of Service Actions according to the parameters `queue.group.max.time` and `queue.group.max.num`. It is an optional parameter and if not defined defaults to `false`. This parameter can be also configured for each particular Queue.
- `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 (it will wait indefinitely). This parameter can be also configured for each particular Queue.
- `queue.group.max.num`: Number of elements that form a group if `queue.group.max.time` has not expired. It is an optional parameter and if not defined defaults to 0, which means that there is no limitation on the maximum number. This parameter can be also configured for each particular Queue.
- `queue.max.parallelism`: Maximum number of Service Actions being processed simultaneously. It is an optional parameter and if not defined defaults to 0, which means that the maximum parallelism depends on the Service Action Executors. This parameter can be also configured for each particular Queue.
- `queue.timeout`: Maximum time in milliseconds that a Service Action can be queued. It is an optional parameter and if not defined defaults to 0, which means that there is no timeout. This parameter can be also configured for each particular Queue.
- `queue.threads`: Number of threads consuming from the Queue. It is an optional parameter and if not defined defaults to one consumer. This parameter can be also configured for each particular Queue.
- `queue.scheduler.interval`: Milliseconds between two consecutive reads from queue. It is an optional parameter and if not defined defaults to 0. This parameter can be also configured for each particular Queue.
- `queue.subqueue.parameter`: Name of the parameter where a default name for the Sub-Queues of every Queue has been configured. This means that the configuration file `$SOSA_HOME/properties/sosa-module.properties` must contain a property which name must match the value of this one and which value must be the default name for the Sub-Queues. It is an optional parameter and if not defined defaults to `subqueue_name` (note that by default there is no property called `subqueue_name`, so this won't make effect until that property is added).
- `queue.synchronous`: Accepts the values `true` and `false` and indicates whether the Queue is synchronous (`true`) or not (`false`). If synchronous then the consumer waits for the Service Action to finish before getting another one from the Queue. It is an optional parameter and if not defined defaults to `false`. This parameter can be also configured for each particular Queue.
- `queue.block.on.retry`: Accepts the values `true` and `false` and indicates whether the Queue must be locked until retrying (`true`) or not (`false`). It is an optional parameter and if not defined defaults to `false`. This parameter can be also configured for each particular Queue.
- `queue.subqueue.getter.class`: The Java class containing the business logic to select the best Sub-Queue at a given time. It is an optional parameter and if not defined default to `com.hp.sosa.modules.sosamodule.queues.SubQueueGetterImpl`.
- `queue.subqueue.default`: Name of the default Sub-Queue that will be returned by the `SubQueueGetter` if no other is better.

5.2 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:

- `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 Service Action has failed and will be reinserted, the Service Action will be reinserted into the first place and the place on maximum parallelism will not be release until the Service Action has been re-queued. 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 WFM SAEs to which the load is distributed equally:

```
<Queue className="com.hp.sosa.modules.sosamodule.queues.basic.BasicQueue"
       name="WFM_BASIC_QUEUE">
  <Parameter name="queue.threads" value="5"/>
  <Parameter name="queue.synchronous" value="false"/>
  <Sae medium_load="50" name="WFM_SA_EXECUTOR1"/>
  <Sae medium_load="50" name="WFM_SA_EXECUTOR2"/>
</Queue>
```

5.2.1 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 a given 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:

- `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 Service Action has failed and will be reinserted, the Service Action will be reinserted into the first place and the place on maximum parallelism will not be release until the Service Action has been re-queued. 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:

i. Decide which priority the consumer is going to use.
ii. If there are elements of this priority, get one of them.
iii. 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.
iv. 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:

i. Decide which priority the consumer is going to use.
ii. 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.
iii. 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.
iv. 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.
v. 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 WFM SAEs to which the load is distributed equally:

```
<Queue className=" com.hp.sosa.modules.sosamodule.queues.priority.PriorityQueue"
       name="WFM_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="WFM_SA_EXECUTOR1"/>
  <Sae medium_load="50" name="WFM_SA_EXECUTOR2"/>
</Queue>
```

5.2.2 Dynamic Basic Subqueue

This is the basic implementation of a Queue with Sub-Queues. The Queue will get the Sub-Queue name and if this Sub-Queue is not defined it's created dynamically. In each Sub-Queue 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 Sub-Queue 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 Sub-Queue. Also, the Sub-Queues are available to lock/unlock and open/close. If queue is locked it's locked for all queues. Then a Sub-Queue is unlocked if the queue is unlocked and Sub-Queue unlocked. It is the same to open/close. The group execution is not supported in this type of queue.
To get the Sub-Queue name a class has to be defined. The default class is

```
    com.hp.sosa.modules.sosamodule.queues.dynamicsubqueue.SubQueueGetter
```

If you want to create another you have to extend this class an overwrite the method

```
    public String getSubQueue(String ssid)
```

If this class returns an empty Sub-Queue name, the service will add the default Sub-Queue. It is possible to define the extended class into the catalog through the field called `subqueue assign class`.
The accepted configuration parameters are:

- `queue.threads`: Number of threads consuming from the queue. Default value is one consumer.
- `queue.scheduler.interval`: Milliseconds between two consecutive reads from the queue. Defaults to 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 a SAE.
- `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 Service Action has failed and will be reinserted, the Service Action will be reinserted into the first place and the place on maximum parallelism will not be release until the Service Action has been re-queued. Defaults to `false`.
- `queue.subqueue.max.parallelism`: Default sub-queue maximum 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 maximum time that a sub-queue not defined can live (default value is 0, which implies infinite).
- `queue.subqueue.nameN`: Name of the sub-queue *N*, where *N* must be an increasing number starting with 0.
- `queue.subqueue.max.parallelismN`: Maximum parallelism for the sub-queue *N*, where N must be an increasing number starting with 0.
- `queue.subqueue.error.subqueue.notfound`: Accepts `values` true and `false` and indicates whether an error has to be notified when the requested sub-queue is not found (`true`) or not (`false`). Defaults to `false`.

5.2.3 Dynamic Priority Subqueue

This is the priority implementation of a Queue composed by Sub-Queues. The Queue looks for the requested Sub-Queue using the Sub-Queue name and it will dynamically create it if no Sub-Queue matches the name. Once the Sub-Queue has been selected, the priority is implemented following the FIFO (First In, First Out) policy: elements are appended at the end of the list as they arrive, and the Sub-Queue consumer gets one by one the first elements of the list. The consumers will find the next Sub-Queue available with elements to execute. There are two levels of maximum of parallelism: the global maximum elements to execute and the maximum parallelism allowed for each Sub-Queue. Also, the Sub-Queues are available to lock/unlock and open/close. If a Queue is locked then it becomes locked for every Sub-

Queue. A Sub-Queue is unlocked when both the Queue and Sub-Queue are unlocked. It is the 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 Sub-Queue name a class has to be defined. The default class is

```
com.hp.sosa.modules.sosamodule.queues.dynamicsubqueue.SubQueueGetter
```

If you want to create another you have to extend this class and overwrite the method

```
public String getSubQueue(String ssid)
```

If this class returns an empty Sub-Queue name then the service will use the default Sub-Queue. It is possible to define the extended class into the catalog using the field `subqueue assign class`.
The accepted configuration parameters are:

- `queue.threads`: Number of threads consuming from the queue. Default value is one consumer.
- `queue.scheduler.interval`: Milliseconds between two consecutive reads from the queue. Defaults to 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 a SAE.
- `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 Service Action has failed and will be reinserted, the Service Action will be reinserted into the first place and the place on maximum parallelism will not be release until the Service Action has been re-queued. Defaults to `false`.
- `queue.subqueue.max.parallelism`: Default sub-queue maximum 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 maximum time that a sub-queue not defined can live (default value is 0, which implies infinite).
- `queue.subqueue.nameN`: Name of the Sub-Queue *N*, where *N* must be an increasing number starting with 0.
- `queue.subqueue.max.parallelismN`: Maximum parallelism for the Sub-Queue *N*, where N must be an increasing number starting with 0.

5.2.4 Service Action Executor Selector

By default all Queues use the class

```
com.hp.sosa.modules.sosamodule.queues.ServiceActionExecutorSelector
```

to decide which SAE will be used to execute each Service Action.

5.2.5 Adaptive Service Action Executor Selector

SOSA provides another SAE Selector called Adaptive SAE Selector which is implemented by the Java class

```
com.hp.sosa.modules.sosamodule.queues.AdaptiveServiceActionExecutorSelector
```

This SAE Selector, instead of granting an equitable distribution among the different SAEs, forces an asymmetric distribution among them based on the current number of running jobs in each SAE and granting that the probability of assigning a new job to a SAE is inversely proportional to the number of running jobs in each SAE. In other words, the new job will be assigned to the SAE that supports the lowest load, but taking in mind the configured rate-load for each SAE as it is made in the default SAE Selector as well.

The main idea is that this Adaptive SAE Selector will manage the jobs assignation in a consistent way, assuming eventual connectivity problems or asymmetric performances in the different SAEs.
The usage of this SAE Selector requires that the next configuration is set into the file `$SOSA_HOME/properties/sosa-module.properties`:

- Configure this SAE Selector as the one to be used updating the value of the property called `queue.serviceactionexecutor.selector`:

```
queue.serviceactionexecutor.selector =
   com.hp.sosa.modules.sosamodule.queues.AdaptativeServiceActionExecutorSelector
```

- Add a parameter called `queue.serviceactionexecutor.selector.adaptative.interval` and assign to it the value 20000 (milliseconds). This parameter defines the minimum interval (in milliseconds) required until the internal management of the rate-load on each SAE is updated. This is an optional parameter and if not defined its default value will be 0, this meaning that the rate-load will only be measured when the Selector is initialized.
- Additionally another parameter called `queue.serviceactionexecutor.selector.adaptative.exponentialfit` can also be defined to intensify the differences obtained among the number of running jobs on each SAE, hence causing that the distribution is made in less time. This is an optional parameter and its default value is the exponent 2, this meaning that the obtained number of running jobs on each SAE will be squared (i.e. multiplied by itself). The higher the value is configured, the quicker the job assignation will be balanced. Note that if the value 0 is configured then the number of running jobs on each SAE is not taken in mind since the result of any value with the exponent 0 is 1, so the rate-load on each SAE will always be 1 and the rate-load on every SAE will be considered the same.

5.2.6 Customizing the Service Action Executor Selector

It is possible to develop a customized SAE Selector and extend the functionality provided by SOSA's default SAE Selector. This might be required due to a particular scenario or to the customer's requirements. To do so, the next steps must be followed:

i. Edit the file `$SOSA_CONFIG/sosa_conf.xml`, look for the Queue(s) to which the customized SAE Selector will apply and set it through the attribute `saeSelectorClassName` of the tag `<Queue>`.
ii. Write the Java class and deploy it below `$SOSA_HOME/lib/queue`.

Note: The directory where customized Java SAE Selectors must be deployed is configurable so it can be different than `$SOSA_HOME/lib/queue` as it is explained in the section related to Queues Configuration.

Example:
A new implementation called `CustomSAESelector` is being defined and applied to the Queue called `BasicQueue`. The first thing required is to define and configure it in the file `$SOSA_CONFIG/sosa_conf.xml` as it is shown below:

```
<Queue className="com.hp.sosa.modules.sosamodule.queues.basic.BasicQueue"
       name="WFM_BASIC_QUEUE" saeSelectorClassName=”mypackage.CustomSAESelector”>
  <Parameter name="queue.threads" value="5"/>
  <Parameter name="queue.synchronous" value="false"/>
  <Sae medium_load="50" name="WFM_SA_EXECUTOR1"/>
  <Sae medium_load="50" name="WFM_SA_EXECUTOR2"/>
</Queue>
```

The next step is to write the source code of the customized SAE Selector, extending the Java class which implements the default SAE Selector

```
   public class CustomSAEASelector extends ServiceActionExecutorSelector
```

and implement the required methods with the new features:

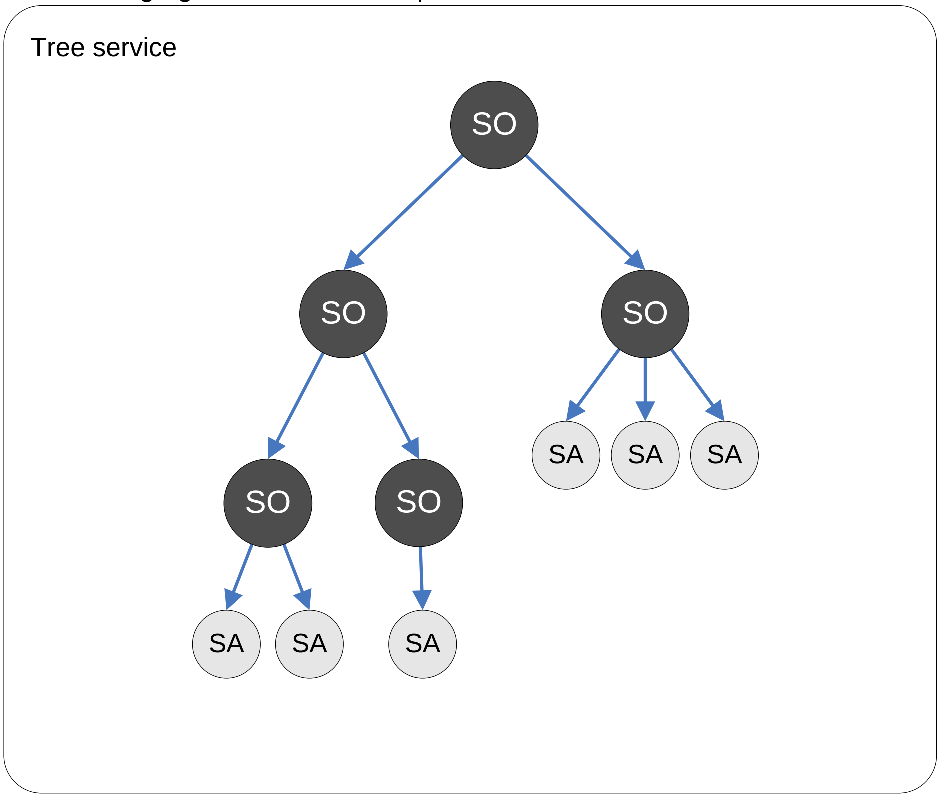
- `public void init(Sae[] saes, Map parameters)`: initialize the custom SAE Selector and configure it with the initial parameters.
- `public ServiceActionExecutor getServiceActionExecutor()`: return the next SAE to use.
- `public synchronized void addServiceActionExecutor(Sae sae)`: add a new SAE to this selector.
- `public synchronized void removeServiceActionExecutor(String name)`: remove the given SAE from this selector.
- `public Sae[] getSaes()`: gets the existing SAEs.

6 Service Catalog

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

6.1 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:



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.

6.2 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 tag `<Catalog>`, 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 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>
```

6.3 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.

6.3.1 Service Orders

The Service Order configuration requires the next mandatory parameters:

- `service_order_name`: The name of the Service Order. This is a required parameter and along with the `service_name` and the `operation` uniquely identifies the Service Order.
- `service_name`: The name of the Service. It is a required parameter and along with the `service_order_name` and the `operation` uniquely identifies the Service Order.
- `operation`: The operation name. It is just a name to identify the Service Order, so it doesn't need to reference any existing operation concept. It is a required parameter and along with the `service_order_name` and the `service_name` uniquely identifies the Service Order.
- `timeout`: The time, in milliseconds, before a timeout occurs.

Additionally, the next optional parameters can be also configured:

- `persistable`: Accepts the values true and false and indicates whether the Service Order must be persisted (true) or not (false). If a Service Order is marked as persistent, the tree of services built from the Service Order will be stored in the persistent database during its execution.
- `processing_mode`: Accepts the values `serial` and `parallel` and indicates whether the children of the Service Order have to 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 the children will be executed at the same time.
- `state`: The state of the service order, which can be `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.

6.3.2 Service Actions

The Service Action configuration requires the next mandatory parameters:

- `service_action_name`: The name of the Service Action. This is a required parameter and along with the `service_name` and the `operation` uniquely identifies the Service Action.
- `service_name`: The name of the Service. It is a required parameter and along with the `service_action_name` and the `operation` uniquely identifies the Service Action.
- `operation`: The operation name. It is just a name to identify the Service Action, so it doesn't need to reference any existing operation. It is a required parameter and along with the `service_action_name` and the `service_name` uniquely identifies the Service Action.
- `wf_name`: The name of the work flow that will be started up when this Service Action is executed.
- `work_queue`: The name of the Queue which will handle the Service Action.
- `timeout`: The time, in milliseconds, before a timeout occurs.

Optionally, the next parameters are also accepted:

- `wf_name_undo`: The name of the work flow that will be started up in case of error to undo the operations performed by the work flow set in `wf_name`.
- `extra_info`: Eventual extra data for the Service Action Processor.
- `state`: The state of the Service Order, which can be one of `enable`, `disable`, `reject` or `simulate`.
- `num_retry`: The number of times to retry the operation in case of error.

- `retry_interval`: The time, in milliseconds, to wait before retrying.
- `user_mapper_class`: The name of the Java class used to map parameters among Service Actions of the same Service Order. This class must implement the interface
  ```
  com.hp.sosa.modules.sosamodule.elements.UserMapperClass
  ```
  provided with SOSA and hence implement the methods
  ```
  public void beforeProcessing(ServiceAction sa)
  public void afterProcessing(ServiceAction sa)
  ```
  This class must be stored below `$SOSA_HOME/lib/user`.

Note: The directory `$SOSA_HOME/lib/user` is configurable. See the description of the parameter `user.class.directory` in `sosa-module.properties`.

- `subqueue_parameter`: The name of the parameter used to set the name of the Sub-Queue that will handle the Service Action.
- `subqueue_assign_class`: The Java class used to assign the best Sub-Queue at a given moment. This class must be one of the available `SubQueueGetter` classes available in the system. See the section related to Queues for further information.
- `error_codes_retry`: A comma separated list of error codes that will force in case of error to retry the Service Action.
- `num_retry_list`: A comma separated list with the number of times to retry the Service Action for each configured error code (see `error_codes_retry` above). The length of this list must be the same as the one in `error_codes_retry`.
- `end_action_class`: The name of the Java class used to map parameters among Service Actions of the same Service Order. This class must implement the interface
  ```
  com.hp.sosa.modules.sosamodule.elements.UserEndActionClass
  ```
  provided with SOSA and hence implement the methods
  ```
  public boolean doRetry(ServiceAction sa)
  ```
  This class must be stored below `$SOSA_HOME/lib/user`.

Note: The directory `$SOSA_HOME/lib/user` is configurable. See the description of the parameter `user.class.directory` in `sosa-module.properties`.

- `on_error_wait_operator`: Accepts the values `true` and `false` and indicates whether a manual interaction is required in case of error (`true`) or not (`false`).
- `queue_parameters`: A comma separated list containing eventual parameters required by the Queue. They must be set this way:
  ```
  param1=value1,param2=value2,…
  ```
- `use_sa_unique_name`: Use the 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`: Accepts the values true and false and indicates whether the Service Order to which the Service Action belongs has to return immediately to its parent (or the Protocol Adapter if it is the root Service Order) with an OK code and continue executing as usual afterwards. The return code and its description can be checked in the history or persistence views.

6.3.3 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.

Input and Output mappings can be configured in `$SOSA_HOME/properties/sosa-module.properties`:

- `inputmapping.space.input`: Defaults to I
- `inputmapping.space.context`: Defaults to C
- `inputmapping.space.output`: Defaults to O
- `outputmapping.space.input`: Defaults to I
- `outputmapping.space.context`: Defaults to C
- `outputmapping.space.results`: Defaults to R
- `outputmapping.space.output`: Defaults to O

6.3.4 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.

6.3.5 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.

6.3.5.1 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`.

6.3.5.2 Request Semantics

A dynamic order request consists of the following elements:

a) Service Request

This is the root element `<serviceRequest>`. It must contain a child tag `<services>` and it may also contain an optional `<header>` element. It doesn't accept attributes.

b) Header

This is an optional child of the `<serviceRequest>` element. It 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.

6.3.5.3 Service Request Example

Here is an example of a JDBC catalog configuration:

```
<?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>
```

6.4 Localization

SOSA, and particularly the Service Catalog, can be localized. The available localized texts must be configured in
`$SOSA_HOME/properties/sosa-module.properties`
Prefixes employed in the Service Catalog to reference not persisting elements are:

- `so.tree.id.prefix.nopersistable`: The prefix applied to the Catalog Tree when it is not persisted. It defaults to `tree-nopers`.
- `so.job.id.prefix.nopersistable`: The prefix applied to the work flows that are not persisted. It defaults to `job-nopers`.
- `so.id.prefix.nopersistable`: The prefix applied to Service Orders that are not persisted. It defaults to `so-nopers`.
- `sa.id.prefix.nopersistable`: The prefix applied to Service Actions that are not persisted. It defaults to `sa-nopers`.

The state names in which Service Orders and Service Actions can be found while they are working are:

- `status.created`: Defaults to `CREATED`.

- `status.built`: Defaults to `BUILDED`.
- `status.scheduled`: Defaults to `SCHEDULED`.
- `status.pause`: Defaults to `PAUSE`.
- `status.processed`: Defaults to `PROCESSED`.
- `status.wait.child`: Defaults to `WAIT_CHILD`.
- `status.returned`: Defaults to `RETURNED`.
- `status.processing`: Defaults to `PROCESSING`.
- `status.enqueued`: Defaults to `ENQUEUED`.
- `status.error`: Defaults to `ERROR`.

The state names in which Service Orders and Service Actions can be found while they are not working are:

- `status.enable`: Defaults to `enable`.
- `status.disable`: Defaults to `disable`.
- `status.reject`: Defaults to `reject`.

The names of the processing modes are:

- `mode.serial`: Defaults to `serial`.
- `mode.parallel`: Defaults to `parallel`.

7 Protocol Adapters

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

7.1 Configuration

The configuration parameters described in this section are general to all the Protocol Adapters defined in the system and hence they will apply to all of them.
These common configuration parameters must be set in `$SOSA_HOME/properties/sosa-module.properties`.

- `protocol.adapter.class.directory`: The directory in the file system where the Java classes implementing the Protocol Adapters will be stored. It is an optional parameter and if not defined defaults to `$SOSA_HOME/lib/protocoladapter`.
- `protocol.adapter.return.wait.retry`: Time, in milliseconds, at which the services handled by the Protocol Adapter have to be awakened. It is an optional parameter and if not defined defaults to 5000 milliseconds (i.e. 5 seconds).

7.2 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.

7.3 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 client, this means the client should publish the RMI or web service 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.

7.3.1 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: Name of the host where SOSA is running.
- Port: 1119

7.3.2 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
ws.secured	True will make the protocol adapter to listen in https.
ws.secured.keystore	Mandatory it ws.secured=true. Keystore used to store

	the SSL certificate.
ws.secured.password	Mandatory it ws.secured=true. Password of the SSL certificate
ws.secured.keyPassword	Mandatory it ws.secured=true. Password of the keystore.
ws.secured.protocol	Only valid if ws.secured=true. SSL Protocol. Default value is "TLS".
ws.secured.algorithm	Only valid if ws.secured=true. Algorithm of the certificate. Default value is "SunX509".
ws.secured.keystoreType	Only valid if ws.secured=true. Type of the keystore. Default value is "JKS".
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.

7.3.2.1 Multiple instances of Web Service Protocol Adapter

If more than one web services protocol adapter is desired, some extra manual steps are requested in order associate each protocol adapter with the web service instance. This can be done by defining new services in the provided axis 1.4 or by duplicating the entire web service.

1. Define all the protocol adapters in the sosa_config.xml file
2. Creating the new services. This can be done by adding service to the existent axis web application or by duplicating the axis.
   a. For adding new service you have to edit
      `$ACTIVATOR_OPT\EP\SOSA\webapps\axis1.4\WEB-INF\server-config.wsdd`
      and:
      i. Copy the entire service named `WSPAClientService` until you have the same number of services than WS Protocol Adapters
      ii. Name them with different literals
      iii. Rename the parameter `className` to a different class
          E.g.
          `com.hp.sosa.modules.sosamodule.protocoladapters.ws.WSPAClntSrvSOAPSkln_1`
          for each entry.
   b. For duplicating the entire web service you have to copy the entire directory
      `$ACTIVATOR_OPT\EP\SOSA\webapps\axis1.4`
      to
      `$ACTIVATOR_OPT\EP\SOSA\webapps\<name>`
      This has to be done for every different Protocol Adapter. Also, you have to edit
      `$ACTIVATOR_OPT\EP\SOSA\webapps\<name>\WEB-INF\server-config.wsdd`
      and:
      i. Change the references from axis1.4 directory to <name> directory
      ii. In service named `WSPAClientService`, rename the parameter `className` to a different class
          E.g.
          `com.hp.sosa.modules.sosamodule.protocoladapters.ws.WSPAClntSrvSOAPSkln_1`
          for each entry.

   Do not forget to set the `ws.webapp.path` parameter in the Protocol Adapter definition (in the file `$SOSA_CONFIG/sosa_conf.xml`) to the new web application route.
3. Create all the classes defined previously in the `className` parameter. In this example this is reduced to
   `com.hp.sosa.modules.sosamodule.protocoladapters.ws.WSPAClientServiceSOAPSkeleton_1`
   They must extend from
   `com.hp.sosa.modules.sosamodule.protocoladapters.ws.WSPAClientServiceSOAPSkeleton`

and, in the constructor, assign the name of the Protocol Adapter associated to, like:

```
public WSPAClntSrvSOAPSkln_1() {
  super("PROTOCOL_ADAPTER_1");
}
```

7.3.3 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.

7.3.4 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 web service the client must implement next WSDL.
Using the Protocol Adapter clients, customers can develop theirs owns clients by using these methods.

7.3.5 Creating a 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 named `NewExampleProtocolAdapter`. The first step is extend the class

```
com.hp.sosa.modules.sosamodule.protocoladapters.ProtocolAdapter
```

Like:

```
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. At this point it is required to remove the configured Protocol Adapter or finish the connections.
- `public String getErrorMessage()`: in case the Protocol Adapter detect some error with the external system return an 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 databases it is possible to define two listeners, one for each database. Usually, it is more common to define more Protocol Adapter instances to avoid using listeners. This method returns the name of the available listeners. Return {} is there aren't.

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

- `public boolean haveToInsertService(String ssid)`: when SOSA start and detects the root Service Order 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 SAE. This method will be called by SOSA when start and before to send any Service Action 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 SAE. 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
{
  String ssid;
  if (this.isRunning()) {
    log.debug("Starting asynchronous service order”);
    ssid = super.getNewBuildedServiceOrder(type, service, action, inputParams, user);
    super.insertServiceOrder(ssid);
    return ssid;
  }
  log.info("Start up of asynchronous service order failed -- the PA is not running");
  throw new SosaException("Cannot start the service order, the PA 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)`.

8 Persistence

SOSA 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 were 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.

8.1 Hibernate Persistence

SOSA 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
`$SOSA_CONF/hbm_persistence.cfg.xml`.

Note: The name and location of the XML file used to configure Hibernate is also configurable. This can be modified editing the file `$SOSA_HOME/properties/sosa-module.properties` and defining the property `persistence.hibernate.config.file` with the reference to the new file. As it can be supposed, the default value of this property is

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

If this persistence is configured into SOSA then the file
`$EP_WAR/properties/hbm_persistence.properties`

needs to contain the next line:

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

8.2 File mixed Hibernate Persistence

SOSA is provided with persistence for high performance solutions. This persistence first saved the order in file because is much faster and in case the order spends a lot of time inside of SOSA change the persistence to hibernate. It is not possible to search into the administration web this kind of services.
To configure Hibernate on this persistence SOSA uses the XML configuration file

`$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 `MixedServiceOrder.hmb.xml` and `MixedServiceAction.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.MixedFileHibernate
SosaPersistence">
   <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.

If this persistence is configured into SOSA then the file

```
   $EP_WAR/properties/hbm_persistence.properties
```

needs to contain the next line:

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

9 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 their 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.

9.1 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:

```
<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, …
- `history.hibernate.cleaner.max.time.in.days`: every day, SOSA will clean the history entries that are older, in days, than this value. A value of 0 means that not deletion will be made. The default value is 30.
- `history.hibernate.vendor:` database vendor. It can be "Oracle" or "EnterpriseDB". By default is Oracle.

There are, also, some advanced parameters that are better not to modify unless a very specific need:

- `history.hibernate.number.threads.consumers`: Number of threads used to store services in the DB. The default value is 8.

9.2 Partition history tables

In case the partition table needs to be partitioned the script `history_partition_creation.sql` has to be executed. Note that this operation requires being very careful as all the data in the historical tables 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.
- `history.hibernate.partition.tablespace1:` Table space for the partitions that are created on Mondays. It is also possible to configure the rest of the days with the parameters tablespace2, tablespace3, etc.

9.3 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,
Also, it's also possible to force use only one Map. In that case, next are the configuration examples:

```
<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 `SOSA.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 `SOSA.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.

9.4 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
```

9.5 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.

9.6 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, so all changes into SA will be saved after this method.

9.7 TPS (Transaction Per Second) Manager

The TPS manager saves the number of transaction per second created and returned. This information is saved in the database and into the CSV (Comma Separated Values) formatted file `$SOSA/tps/year.csv` . This manager can save the "root SO" (only one transaction per tree) or the number of SO which contains SA (can save more transaction per tree). The first option is recommended for solutions that only a tree contains a transaction. In the second case, it is recommended for solutions that a tree can contains more than one transaction and a transaction is a SO with SA children.
In case the solution has more than one SOSA or SOSA is configured in high availability, it is mandatory to configure the parameter `sosa.instance.name` in sosa.xml SosaModule. See next, as example:

```
<Modules>
   <Module name="sosaModule" className="com.hp.sosa.modules.sosamodule.SosaModule">
      <Parameter name="sosa.conf.file" value="conf/sosa_conf.xml" />
      <Parameter name="jetty.start" value="true" />
      <Parameter name="sosa.instance.name" value="sosa1" />
   </Module>
...
```

```
</Modules>
```

If the parameter `tps.manager.force.add` in the configuration of SOSA module is `true` this manager will be added even if it is not configured in `sosa_conf.xml`. In addition, the module will use the catalog database configuration if it is not provided.
Below is an example of the TPS configuration:

```
<Manager className="com.hp.sosa.modules.sosamodule.managers.tps.TPSManager" name="TPS">
  <Parameter name="tps.db.reload.period" value="600000"/>
  <Parameter name="tps.db.autoreload" value="true"/>
  <Parameter name="tps.db.user" value="userpp"/>
  <Parameter name="tps.db.password" value="userpp"/>
  <Parameter name="tps.db.jdbc.driver" value="oracle.jdbc.driver.OracleDriver"/>
  <Parameter name="tps.db.driver.name" value="jdbc:oracle:thin"/>
  <Parameter name="tps.db.host" value="127.0.0.1"/>
  <Parameter name="tps.db.port" value="1521"/>
  <Parameter name="tps.db.instance" value="HPSA"/>
  <Parameter name="tps.method.is.root" value="true"/>
  <Parameter name="tps.period.minutes" value="1"/>
  <Parameter name="tps.interval.minutes" value="60"/>
</Manager>
```

The TPS manager supports several configuration parameters:

Name	Description
tps.period.minutes	TPS manager will count the number of transactions per second for each period.
tps.interval.minutes	For each interval TPS will save the maximum and average orders per second in this interval
tps.max.years.history	Maximum years of data. By default 5 years. Old data will remove automatically. 0 means that old data has not to be removed.
tps.method.is.root	If true only one transaction per tree will be counted. If false one transaction for each SO with SA children will be counted.
tps.db.pool.name	Name of the database connections pool.
tps.db.host	Hostname of the machine where the database is present.
tps.db.port	Port to connect.
tps.db.instance	Oracle instance which stores the tps.
tps.db.user	Username of the instance used for the tps.
tps.db.password	Password of the username.
tps.db.driver.name	Name of the driver used.
tps.db.jdbc.driver	JDBC driver used.
tps.db.initialsize	Initial size of connections pool.
tps.db.maxactive	Maximum time a connection is active.
tps.db.maxidle	Maximum time a connection is idle.
tps.db.minidle	Minimum time for a connection to be marked as idle.
tps.db.maxwait	Maximum time waiting for a connection.

10 Executing Service Orders

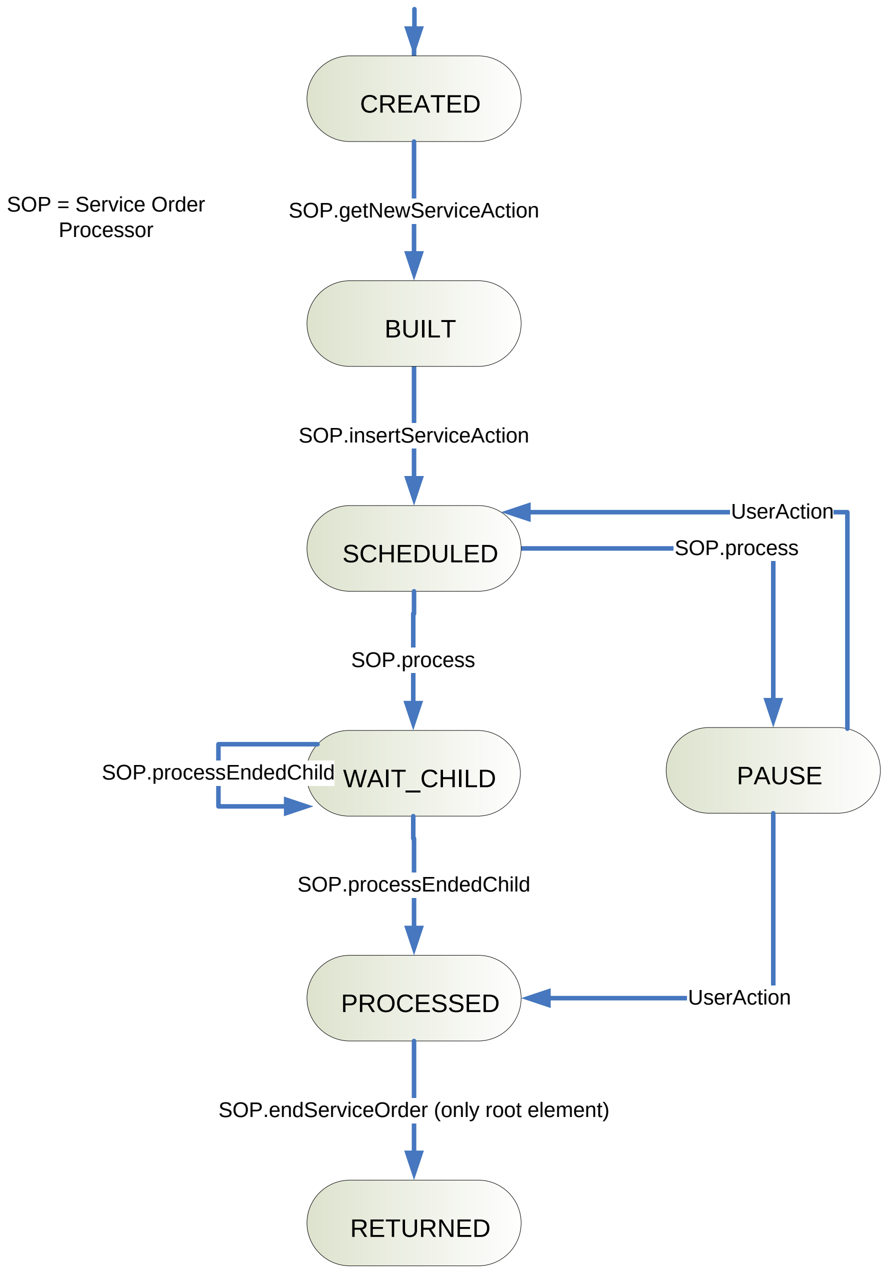
When Service Orders are sent to SOSA to be processed, each service of the tree service goes through different states while being executed.
The execution of Service Orders last from the protocol adapter inserts them into SOSA to SOSA responds the protocol adapter.

10.1 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.

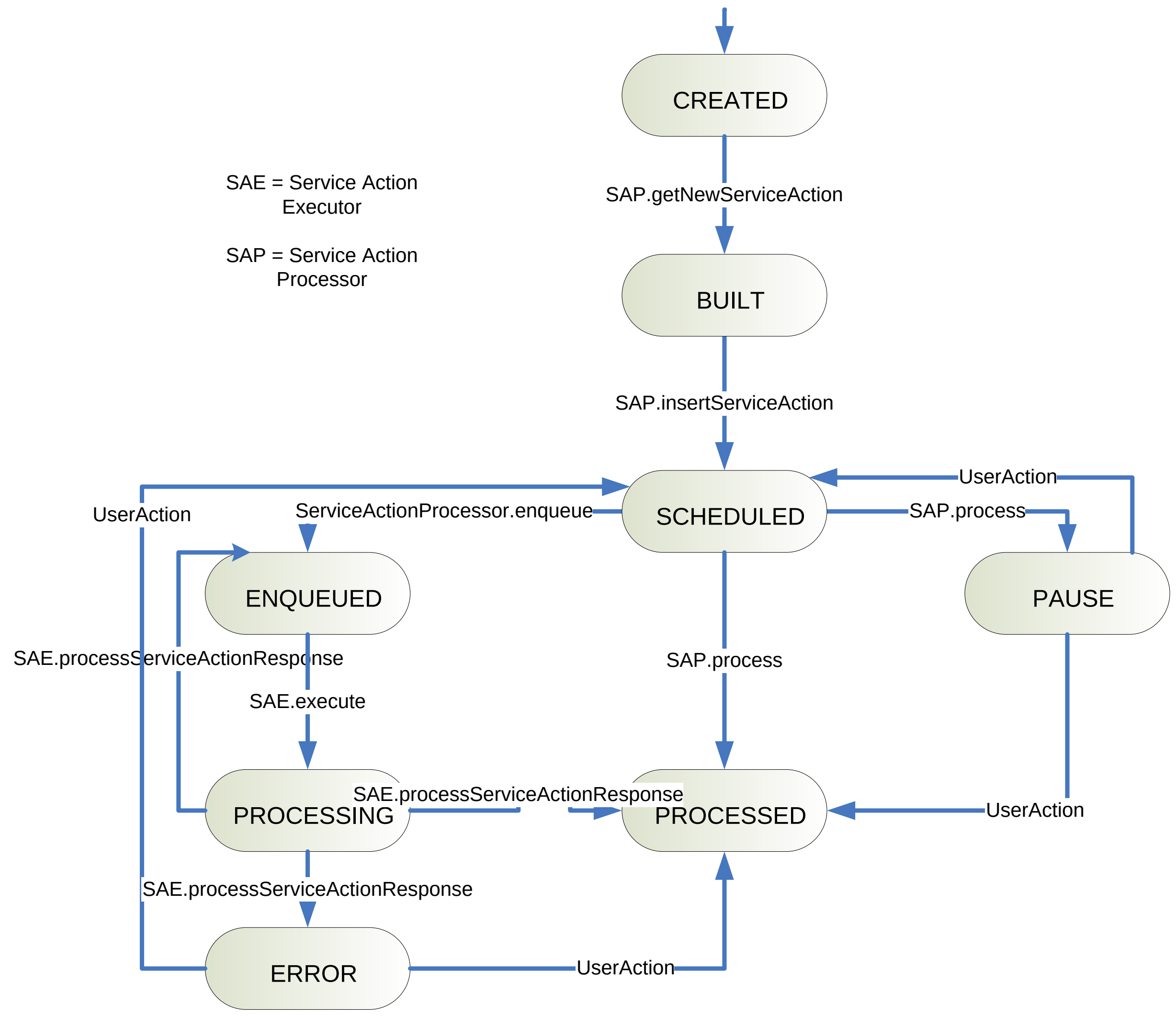
Next figure shows the Service Order state diagram:



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:


SAE = Service Action Executor
SAP = Service Action Processor
CREATED
SAP.getNewServiceAction
BUILT
SAP.insertServiceAction
SCHEDULED
UserAction
SAP.process
PAUSE
UserAction
ServiceActionProcessor.enqueue
ENQUEUED
SAE.processServiceActionResponse
SAE.execute
SAP.process
PROCESSING
SAE.processServiceActionResponse
PROCESSED
UserAction
SAE.processServiceActionResponse
ERROR
UserAction

11 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.

12 Tips and recommendations

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

12.1 Performance tips

If you are planning to use SOSA in a high load scenario you might need to work a bit in the performance. Here are some tips to help improving it:

a) The most critical task in a performance's point of view is the persistence. Also it is a very important feature that we cannot get rid of it in most of the cases. The best option in performance's point of view is to configure the `MixedFileHibernateSosaPersistence`. Have in mind that, if you have more than one hard disk, you can configure `persistence.file.number.harddisk` parameter to speed up the write to disk process.

b) Another important feature is the historic manager. As the services are first stored in the hard disk, if you have more than one hard disk, you can configure `persistence.file.number.harddisk` parameter to speed up the write to disk process. It is very important to remove the old historic entries or, soon or later, the cost of adding new entries to the history tables will be too high. The recommended way is to use Partitions (check Partition history tables). As the drop of the old partitions will be handle by that SOSA module it is wise to turn off the normal history cleaning process by setting `history.hibernate.cleaner.max.time.in.days` parameter to 0.

c) One typical problem that prevents SOSA to serve more orders is to wait to the workflows to end. So when the workflows are time-consuming the best idea is to use the asynchronous approach of SOSA.

12.1.1 Processors

A Processor is an internal component that handles every service started up in SOSA until it is finished, and it is responsible of the service state updates, so this component will be the one which will notice the Protocol Adapter when the service is finished.
The Processor behavior is based on consumers, which are working units that parallelize their tasks whenever it is possible.
Processors are not public entities, but they accept some configuration parameters that may have an impact on the system performance. There are Service Order Processors as well as Service Action Processors, and both of them can be configured in `$SOSA_HOME/properties/sosa-module.properties`.

12.1.1.1 Service Order Processors

All the configuration parameters described in this section are optional.

- `service.order.processor.process.scheduler.consumers`: The number of consumers in charge of processing services that will be ready for each Service Order Scheduler. It defaults to 10.
- `service.order.processor.process.scheduler.consumers.reusethread`: Accepts the values `true` and `false` and indicates whether the consumers in charge of processing services will be kept alive even when their current task has been finished (`true`) or if they have to be destroyed and re-created every time that a new task needs to be processed (`false`). It defaults to `true`.

- `service.order.processor.ended.scheduler.consumers`: The number of consumers in charge of finished services that will be ready for each Service Order Scheduler. It defaults to 10.
- `service.order.processor.ended.scheduler.consumers.reusethread`: Accepts the values `true` and `false` and indicates whether the consumers in charge of finished services will be kept alive even when their current task has been finished (`true`) or if they have to be destroyed and re-created every time that a new task needs to be processed (`false`). It defaults to `true`.
- `service.order.processor.timeout.scheduler.consumers`: The number of consumers in charge of timed out services that will be ready for each Service Order Scheduler. It defaults to 10.
- `service.order.processor.timeout.scheduler.consumers.reusethread`: Accepts the values `true` and `false` and indicates whether the consumers in charge of timed out services will be kept alive even when their current task has been finished (`true`) or if they have to be destroyed and re-created every time that a new task needs to be processed (`false`). It defaults to `true`.
- `service.order.processor.return.pa.scheduler.consumers`: The number of consumers in charge of noticing the Protocol Adapter when a service is finished that will be ready for each Service Order Scheduler. It defaults to 10.
- `service.order.processor.return.pa.scheduler.consumers.reusethread`: Accepts the values `true` and `false` and indicates whether the consumers in charge of noticing the Protocol Adapters will be kept alive even when their current task has been finished (`true`) or if they have to be destroyed and re-created every time that a new task needs to be processed (`false`). It defaults to `true`.
- `service.order.processor.propagate.input.from.parent.to.child`: Accepts the values `true` and `false` and indicates whether the input from the parent service has to be available for its children (`true`), if any, or not (`false`). It defaults to `true`.
- `service.order.processor.propagate.input.from.child.to.parent`: Accepts the values `true` and `false` and indicates whether the input from the child service has to be available for its parent (`true`), if any, or not (`false`). It defaults to `false`.
- `sosa.code.ok`: The code that can be received from the external system to indicate that this Service Order has finished successfully. The default value is 0.
- `sosa.code.ok.list`: A comma separated list of codes that can be received from the external system to indicate that this Service Order has finished successfully.

12.1.1.2 Service Action Processors

All the configuration parameters described in this section are optional.

- `service.action.processor.process.scheduler.consumers`: The number of consumers in charge of processing services that will be ready for each Service Action Scheduler. It defaults to 10.
- `service.action.processor.process.scheduler.consumers.reusethread`: Accepts the values `true` and `false` and indicates whether the consumers in charge of processing services will be kept alive even when their current task has been finished (`true`) or if they have to be destroyed and re-created every time that a new task needs to be processed (`false`). It defaults to `true`.
- `service.action.processor.ended.scheduler.consumers`: The number of consumers in charge of finished services that will be ready for each Service Action Scheduler. It defaults to 10.
- `service.action.processor.ended.scheduler.consumers.reusethread`: Accepts the values `true` and `false` and indicates whether the consumers in charge of finished services

will be kept alive even when their current task has been finished (`true`) or if they have to be destroyed and re-created every time that a new task needs to be processed (`false`). It defaults to `true`.

- `service.action.processor.timeout.scheduler.consumers`: The number of consumers in charge of timed out services that will be ready for each Service Action Scheduler. It defaults to 10.
- `service.action.processor.timeout.scheduler.consumers.reusethread`: Accepts the values `true` and `false` and indicates whether the consumers in charge of timed out services will be kept alive even when their current task has been finished (`true`) or if they have to be destroyed and re-created every time that a new task needs to be processed (`false`). It defaults to `true`.
- `service.action.processor.return.pa.scheduler.consumers`: The number of consumers in charge of noticing the Protocol Adapter when a service is finished that will be ready for each Service Action Scheduler. It defaults to 10.
- `service.action.processor.return.pa.scheduler.consumers.reusethread`: Accepts the values `true` and `false` and indicates whether the consumers in charge of noticing the Protocol Adapters will be kept alive even when their current task has been finished (`true`) or if they have to be destroyed and re-created every time that a new task needs to be processed (`false`). It defaults to `true`.
- `service.action.processor.propagate.input.from.parent.to.child`: Accepts the values `true` and `false` and indicates whether the input from the parent service has to be available for its children (`true`), if any, or not (`false`). It defaults to `true`.
- `service.action.processor.propagate.input.from.child.to.parent`: Accepts the values `true` and `false` and indicates whether the input from the child service has to be available for its parent (`true`), if any, or not (`false`). It defaults to `false`.

12.1.1.3 Common configuration

The parameters described in this section can be configured in `$SOSA_HOME/properties/sosa-module.properties` and are common for all kind of Processors.

- `sosa.code.ok`: The code that can be received from the external system to indicate that this Service Action has finished successfully. The default value is 0.
- `sosa.code.ok.list`: A comma separated list of codes that can be received from the external system to indicate that this Service Action has finished successfully.
- `sosa.code.reject`: The code that can be received from the external system to indicate that this Service Action has been rejected. The default value is 100.
- `sosa.description.reject`: The description stored in the Service Action when it is rejected.
- `sosa.code.unknown.state`: The code that can be received from the external system to indicate a state that does not match any of the expected ones. The default value is 110.
- `sosa.description.unknown.state`: The description stored in the Service Action when the received state is unknown.
- `sosa.code.changestatus`: The code stored in the Service Action when it cannot be scheduled. It defaults to 130.
- `sosa.description.changestatus`: The description stored in the Service Action when it cannot be scheduled.
- `sosa.code.enqueue.error`: The code stored in the Service Action when it cannot be queued. It defaults to 140.
- `sosa.description.enqueue.error`: The description stored in the Service Action when it cannot be queued.

- `sosa.code.process.error`: The code stored in the Service Action when it cannot be processed by SOSA due to an unspecified internal error. It defaults to 150.
- `sosa.description.process.error`: The description stored in the Service Action when it cannot be processed.
- `sosa.code.notexists.jobid`: The code stored in the Service Action when the job identifier associated to the internal work flow cannot be found (this can happen if the work flow has finished unexpectedly in the external system). It defaults to 151.
- `sosa.description.notexists.jobid`: The description stored in the Service Action when the job identifier of the internal work flow cannot be found.
- `sosa.code.sa.exec.error`: The code stored in the Service Action when the SAE detects a runtime error during the Service Action execution. It defaults to 160.
- `sosa.description.sa.exec.error`: The description stored in the Service Action when the SAE detects a runtime error.
- `sosa.code.sa.exec.error.getsae.conn:` The code stored in the Service Action when the SAE cannot be found. It defaults to 161.
- `sosa.description.sa.exec.error.getsae.conn:` The description stored in the Service Action when the SAE cannot be found.
- `sosa.code.sa.exec.error.getsa:` The code stored in the Service Action when the Service Action cannot be found. It defaults to 162.
- `sosa.description.sa.exec.error.getsa:` The description stored in the Service Action when the Service Action cannot be found.
- `sosa.code.sa.exec.timeout:` The code stored in the Service Action when the SAE finds that the Service Action has been timed out. It defaults to 163.
- `sosa.description.sa.exec.timeout:` The description stored in the Service Action when the SAE finds that the Service Action has been timed out.
- `sosa.code.sa.exec.not.notified:` The code stored in the Service Action when no notification is obtained about the Service Action result. It defaults to 164.
- `sosa.description.sa.exec.not.notified:` The description stored in the Service Action when the Service Action result is not obtained.
- `sosa.code.canceled:` The code stored in the Service Action when the Service Action has been cancelled. It defaults to 190.
- `sosa.description.canceled:` The description stored in the Service Action when the Service Action has been cancelled.
- `sosa.code.timeout:` The code stored in the Service Action when the Service Action Processor finds that the Service Action has been timed out. It defaults to 200.
- `sosa.description.timeout:` The description stored in the Service Action when the Service Action Processor finds that the Service Action has been timed out.
- `sosa.code.restore.error:` The code stored in the Service Action when it cannot be restored. It defaults to 210.
- `sosa.description.restore.error:` The description stored in the Service Action when it cannot be restored.
- `sosa.code.queue.closed:` The code stored in the Service Action when the Queue has been closed during its execution. It defaults to 220.
- `sosa.description.queue.closed:` The description stored in the Service Action when its Queue gets closed.
- `sosa.code.queue.not.exist:` The code stored in the Service Action when its Queue cannot be found. It defaults to 230.
- `sosa.description.queue.not.exist:` The description stored in the Service Action when its Queue cannot be found.

- `sosa.code.subqueue.not.exist`: The code stored in the Service Action when its Sub-Queue cannot be found. It defaults to 240.
- `sosa.description.subqueue.not.exist`: The description stored in the Service Action when its Sub-Queue cannot be found.
- `sosa.code.aborted`: The code stored in the Service Action when it is aborted. It defaults to 250.
- `sosa.description.aborted`: The description stored in the Service Action when it is aborted.
- `sosa.code.subqueue.closed`: The code stored in the Service Action when its Sub-Queue has been closed. It defaults to 260.
- `sosa.description.subqueue.closed`: The description stored in the Service Action when its Sub-Queue has been closed.
- `sosa.code.cannot.process`: The code stored in the Service Action when it cannot be processed. It defaults to 270.

12.2 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 SOSA logging system is configured in `$SOSA_HOME/properties/sosa-4logj.properties`. In that file it is possible to add more appenders, change the log level, etc. It is allowed to modify this file in runtime - new changes will be loaded in a while.

12.3 Get a Service Order or Service Action

Usually is required to load a specific Service Order or Service Action. 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);
```

12.4 Modify a Service Action or Service Order

To modify a Service Action or Service Order 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);
   }
}
```

12.5 Load entire tree order

In some case we need to load the entire tree order to check or to find or check a specific status from an object. This is an example of two methods which 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(
            "Load a Service Order tree form persistence failed. 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;
}
```

12.6 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.

12.7 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.