Corus Interoperability Specification

version 3.0

www.sapia-oss.org

Table of Contents

Revision History

Description	Version	Author	Date
First draft	1.0	Yanick Duchesne	2002
Reviewed for 2.0	2.0	Yanick Duchesne	2004
Added process event support	3.0	Yanick Duchesne	Jan 2014

Introduction

The current document describes Corus' interoperability protocol, which allows external processes to be federated under the responsibility of a Corus server.

Process Life-Cycle

External processes are launched through a Corus server, and are thereafter responsible for signaling their presence to the server at a predefined interval. The protocol described in this document explains the interactions between Corus-activated processes and their Corus server.

Launch

Upon its `exec` command being called, a Corus server will launch a given process, by dynamically calling the executable that launches the process in question. At that point, the Corus server passes the following properties (through -D options) to the executable's command-line:

Property	***Description***
`corus.server.host`	Specifies the host of the Corus server that started the process; this is somewhat redundant, since the host is necessarily the same as the one on which the started process “lives” - Coruss cannot start processes on other machines. Yet, the property is provided for the sake of consistency.
`corus.server.main.port`	The “main” port of the Corus server that is starting the process. This port is the one on which the client console connects. It is also the port that is used to have access to Corus's built-in naming service.
`corus.server.http.port`	The port on which the Corus's `HTTP` service listens. This is the port used by remote processes to send XML messages to their Corus server.
`corus.distribution.name`	The name of the distribution to which the process “belongs”.
`corus.distribution.version`	The version of the distribution to which the process belongs.

Property	*Description*
`user.dir`	This is a standard Java system property which has been kept “as is”; it identifies the root directory of a given distribution (the directory where the distribution was extracted). Subdirectories can be accessed relatively to this root directory. Processes can conveniently resolve directories and files by using this property's value – which is convenient to retreive configuration files.
`corus.process.dir`	This property indicates to the process its dedicated directory, which gives it a “private” access to the file system.This directory is cleaned up by the Corus server once the process has been killed. Processes that need access to the file system in write mode should use this directory (instead of the `user.dir` directory – see above property).
`corus.process.id`	Specifies the unique identifier of the process that is started. This identifier is unique for the Corus that generated it, but not necessarily among multiple Corus servers (indeed, two different Corus servers could generate the same identifier).
`corus.process.poll.interval`	Specifies the interval at which the process should poll its Corus server. This interval is given in **seconds**. Fractions of a second can be specify using a decimal number (i.e. the value “0.5” represents half a second).
`corus.process.status.interval`	Specifies the interval at which the process should send its status to the Corus server. This interval is given in **seconds**. Fractions of a second can be specify using a decimal number (i.e. the value “0.5” represents half a second).
`corus.client.analysis.interval`	Specifies the interval at which the client module should analyses its current state to know if it has to send some commands to the Corus server. This interval is given in **seconds**. Fractions of a second can be specify using a decimal number (i.e. the value “0.5” represents half a second).

The above properties are passed to the executable that launches the process as a space-separated list of options, according to the following syntax – which follows the Java standard used to pass “system” properties to a JVM from the command-line:

```
-DpropertyName1=”propertyValue1” -DpropertyName2=”propertyValue2”...
```

As can be seen, the following rules apply:

- property names are preceded by a “D” identifier, which allows to distinguish the properties from other command-line arguments;
- property names are separated from their value by an “=” character;
- property values are passed between quotes, which allows them to contain whitespaces.

Requests and Responses

Communication between a Corus server and its remote processes follows a request/response pattern, where processes act as requestors and expect a response in return; the expected response is one of the following:

- A list of pending commands targeted at the requesting process;
- an error;
- a simple acknowledgement if there were no commands in the process' command queue (kept within the Corus server), and if no error occurred.

There are three types of requests that processes emit: polling, status and restart– which are explained below.

Polling

Once a process has been launched, it is responsible for polling its Corus server at the interval specified by the `corus.process.poll.interval` property. The Corus server internally keeps track of all processes launched through it and will detect the ones that do not respect their polling “contract”; the Corus server will try to terminate a given delinquant process by:

- sending a shutdown message to it ;
- if the above does not succeed, calling the OS' “kill” command – if this applies.

Once a process as been forcefully shutdown, it is automatically restarted – if the process' corresponding distribution configuration specifies so.

Status

At the interval specified by the `corus.process.status.interval`, the remote process must provide its status. The status is a message that gives information about the internal state of the process and, potentially, the application(s) it holds. This status can then be accessed in a centralized fashion through the Corus server's client console, or programmatically by connecting to the Corus server.

If the status and polling intervals are the same, a remote process can send its status as part of its poll message (which, in this case, will also contain the status message).

Process-Initiated Restarts

A process can ask its Corus to restart it. In such a case, the process sends to its Corus a message requesting the restart to occur. The Corus will from then on proceed as if the request had come from a user (through the client console): it will send a shutdown message to the process and restart it with the same parameters as before.

Process Event

A process can receive arbitrary events from a its Corus node: such events are application-specific, and identified by a so-called “event type”. They

are represented as maps, holding name-value string bindings.

Process events are used to send custom signals to processes managed by Corus. It is part of logic defined at the application level to determine what needs to be done when receiving such events – Corus serves as a transport channel only.

Termination

Upon receiving a shutdown message (either user-initiated, automatically from its Corus server, or after it has itself requested it), a remote process is responsible for cleanly shutting down, releasing all resources it currently holds. As part of the shutdown sub-protocol, it is expected that remote processes notify their Corus server about the shutdown completion. For each process that thus notifies its Corus server, the latter will remove its internal reference on the remote process – if the process does not notify its Corus about its shutdown completion, the Corus will resort to an OS “kill”.

Protocol

As was briefly explained, Corus' interoperability relies on a request/response scheme, where remote processes act as requestors.

Generalities

Communication between remote processes and Corus servers is insured by SOAP over HTTP. The general format of request messages must respect the following rules:

- Requests specify a mandatory SOAP header;
- requests must contain a body that encapsulates child elements that correspond to polling and/or status command.

In turn, responses obey the following rules:

- Responses specify a mandatory SOAP header;
- any error following a request is returned to the remote process in the form of a SOAP fault;
- if no error occurs, the reponse's SOAP body contains the list of pending commands targeted at the remote process;
- if no command is pending, then a simple acknowledgement element is given as part of the SOAP body.

Message Format

Requests and responses follow the SOAP 1.1 specification. The messages per say are contained within the SOAP body. A message can contain one to many command(s). The latter are directly specified as children of the SOAP body. Each command within a message must be assigned a unique identifier by the sending party.

Namespace

All Corus-specific elements pertaining to interoperability are associated to the following URI:

```
xmlns:CORUS-IOP="http://schemas.sapia.org/Corus/interoperability/"
```

Requests

Header

All requests must specify a mandatory SOAP header, which itself holds a `Process` element, as the example below demonstrates:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
     <CORUS-IOP:Process xmlns:CORUS-IOP="http://schemas.sapia.org/Corus/interoperability/"
                        CorusPid="{CorusProcessIdentifier}"
                        requestId="{requestIdentifier}" />
  </SOAP-ENV:Header>

  <SOAP-ENV:Body>
    ... message here ...
  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Element	*Attribute*	*Description*	*Mandatory*	*Children*
Process		Encapsulates information that allows to uniquely identify the requesting process.	Yes	-
	CorusPid	The process' identifier – assigned by the process' Corus server. Corresponds to the `corus.process.id` property.	Yes	-
	requestId	The request ID is an arbitrary identifier that the requesting process must assign to the request message. This identifier is sent back to the process in the request's corresponding response. This can be useful for debugging purposes (for example by allowing to figure out which request belongs to which response in log files).	Yes	-

Messages

Request specify their message in the SOAP body. A request message may contain more than one command per body.

Poll

A remote process must poll its Corus server at the interval specified to it by

its `corus.process.poll.interval` property. A “poll” signals to the Corus server that the process is functioning properly; it is similar in goal to a heartbeat.

A poll message typically has a single empty poll element, But it can also encapsulate an additional status element – see further below.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
      ... (see header spec further above) ...
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <CORUS-IOP:Poll xmlns:CORUS-IOP="http://schemas.sapia.org/Corus/interoperability/"
                        commandId="{commandIdentifier}" />
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Element	*Attribute*	*Description*	*Mandatory*	*Children*
Poll		Identifies the “poll” command.	Yes	-
	commandId	The command identifier.	Yes	-

A status command allows a remote process to provide information about its state – and the state of the applications it holds. The status command is not mandatory: a Corus server does not require that a process provides status information. A process sends its status at the interval specified by its `corus.process.status.interval` property. To spare network resources, if this interval and the polling interval are somewhat similar, a process can send both status and poll messages in the same request. If such is the case, the request must contain both the poll and status information as sibling XML elements. The examples below demonstrate both uses.

Status

A status command:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
      ... (see header spec further above) ...
    </SOAP-ENV:Header>

    <SOAP-ENV:Body>
        <CORUS-IOP:Status xmlns:CORUS-IOP="http://schemas.sapia.org/Corus/interoperability/"
                          commandId="{commandIdentifier}">
            <CORUS-IOP:Context name="someTopic">
                <CORUS-IOP:Param name=”item1" value="item1_value" />
                <CORUS-IOP:Param name="item2" value="item2_value" />
            </CORUS-IOP:Context>
            <CORUS-IOP:Context name="someOtherTopic">
                <CORUS-IOP:Param name="item1" value="item1_value" />
                <CORUS-IOP:Param name="item2" value="item2_value" />
            </CORUS-IOP:Context>
        </CORUS-IOP:Status>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Combined status and poll commands:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
      ... (see header spec further above) ...
    </SOAP-ENV:Header>

    <SOAP-ENV:Body>
        <CORUS-IOP:Poll xmlns:CORUS-IOP="http://schemas.sapia.org/Corus/interoperability/"
                        commandId="{commandIdentifier}">

        <CORUS-IOP:Status xmlns:CORUS-IOP="http://schemas.sapia.org/Corus/interoperability/"
                          commandId="{commandIdentifier}">
            <CORUS-IOP:Context name="someTopic">
                <CORUS-IOP:Param name="item1" value="item1_value" />
                <CORUS-IOP:Param name="item2" value="item2_value" />
            </CORUS-IOP:Context>
            <CORUS-IOP:Context name="someOtherTopic">
                <CORUS-IOP:Param name="item1" value="item1_value" />
                <CORUS-IOP:Param name="item2" value="item2_value" />
            </CORUS-IOP:Context>
        </CORUS-IOP:Status>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Element	*Attribute*	*Description*	*Mandatory*	*Children*
Status		Identifies the “status” command.	Yes	Context
	commandId	The command identifier.	Yes	-
Context		Allows to subdivide status information in an ad-hoc fashion. A given topic has a mandatory name attribute, whose value is a character string that is in fact the the topic's identification. This name is arbitrary; the Corus server does not perform any validation on it. The only restriction is that topic names cannot be duplicated.	Yes	Param
	name	The name of the topic	Yes	-
Param		A parameter must have its name and value attributes specified. The only restriction that applies in the case of this element is that there must be a single parameter element with a given name per context.	Yes	-
	name	The unique name of this parameter.	Yes	-
	value	The value of the parameter.	Yes	-

Restart

A remote process can ask its Corus to restart it. In such a case, a restart request is sent to the Corus server. Upon receiving the restart request, the server proceeds to the process' termination (as if that request had come from a user, through the client console) and eventually restarts it. The server replies with an ack to the request.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
      ... (see header spec further above) ...
    </SOAP-ENV:Header>

    <SOAP-ENV:Body>
      <CORUS-IOP:Restart xmlns:CORUS-IOP="http://schemas.sapia.org/Corus/interoperability/"
                         commandId="{commandIdentifier}">
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Element	Attribute	Description	Mandatory	Children
Restart		Sent by a process that “wishes” to be restarted. Upon receiving the request, the Corus server proceeds to the process's termination, and eventually restarts it.	Yes	-
	commandId	The command identifier.	Yes	-

ConfirmShutdown

Following a shutdown command (sent as part of a response to a poll or status request – see futher below for the shutdown message specification), and before terminating, a remote process must confirm to its Corus server that it shut down properly. A shutdown confirmation message has a single empty `ConfirmShutdown` element. Upon receiving the shutdown confirmation, the server replies with an ack, and then proceeds to the cleanup of the process directory. If the latter could not be deleted, the Corus server assumes that the process is still running and holding locks on some resources; it will thus forcefully terminate the process through an OS “kill”, and then proceed (again) to the process directory's cleanup.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
      ... (see header spec further above) ...
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
      <CORUS-IOP:ConfirmShutdown
                xmlns:CORUS-IOP="http://schemas.sapia.org/Corus/interoperability/"
                commandId="{commandIdentifier}">
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Element	Attribute	Description	Mandatory	Children
Confirm Shutdown		Tells to the Corus server that the requesting process has completed its shutdown procedure and will incessantly terminate.	Yes	-
	commandId	The command identifier.	Yes	-

Responses

Header

In a manner analogous to requests, responses must also specify a mandatory SOAP header, which itself holds a Server element, as the example below demonstrates:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
     <CORUS-IOP:Server xmlns:CORUS-IOP="http://schemas.sapia.org/Corus/interoperability/"
                       requestId="{requestIdentifier}"
                       processingTime="{millis}" />
  </SOAP-ENV:Header>

  <SOAP-ENV:Body>
    ... message here ...
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Element	*Attribute*	*Description*	*Mandatory*	*Children*
Server		Encapsulates information that allows to uniquely identify the requesting process.	Yes	-
	requestId	The identifier of the request to which the response corresponds – has the same value as the request identifier which was originally passed as part of the request.	Yes	-
	processing Time	The number of milliseconds that the Corus server took to process the request (does not include transport time).	Yes	-

Messages

Responses encapsulate their message(s) in their SOAP body. Responses can consist of one of the following: an ack, a SOAP fault, one to many commands.

Ack

An ack in sent back to the requesting process if one of the following applies:

- no error occurred processing the request;
- no awaiting command was in the process' command queue within the Corus server.

An ack message is composed of a single, empty `Ack` element. An ack is not considered as a command, and thus does not specify a command identifier.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
      ... (see header spec further above) ...
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <CORUS-IOP:Ack xmlns:CORUS-IOP="http://schemas.sapia.org/Corus/interoperability/" />
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Element	*Attribute*	*Description*	*Mandatory*	*Children*
Ack		Identifies an acknowledgement.	Yes	-

Shutdown

A shutdown command is sent in a response under one of the following conditions:

- the Corus server has detected that the process has not polled it within the predefined interval; it enqueues a shutdown command within the process command queue; the next time the process polls, the command is thus sent as part of the response;
- the process' shutdown has been explicitely ordered by a Corus user (through a “kill” command). In this case, the same as above ensues: the command is enqueued within the Corus server – in the process' command queue – and sent back to the process as part of its next status or poll request – see further above.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
      ... (see header spec further above) ...
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
      <CORUS-IOP:Shutdown xmlns:CORUS-IOP="http://schemas.sapia.org/Corus/interoperability/"
                          commandId="{commandIdentifier}"
                          requestor="{requestorActor}" />
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Element	*Attribute*	*Description*	*Mandatory*	*Children*
Shutdown		Identifies a shutdown command.	Yes	-
	commandId	The command identifier.	Yes	-
	requestor	Identifies the entity that requested that shutdown command. The list of values can be (bu not limites to): “Corus” is Corus shutdown the process because it is unstable, “console” if an administrator used the `kill` command to stop a given process, “process” if the process requested the shutdown using the `restart` command.	Yes	-

ProcessEvent

A process event command is sent in a response when the Corus server has detected that one or more event(s) is/are pending for a given process – upon that process polling the server.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
      ... (see header spec further above) ...
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
      <CORUS-IOP:ProcessEvent xmlns:CORUIOP="http://schemas.sapia.org/Corus/interoperability/"
                          commandId="{commandIdentifier}"
                          type="{type}">

        <CORUS-IOP:Param name="{name1}" value="{value2}" />
        <CORUS-IOP:Param name="{name2}" value="{value2}" />
        …
        <CORUS-IOP:Param name="{nameN}" value="{valueN}" />
      </CORUS-IOP:ProcessEvent>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

As shown above, a `ProcessEvent` element may hold multiple optional bindings, which consist of arbitrary name/value pairs meant for application consumption.

Element	*Attribute*	*Description*	*Mandatory*	*Children*
ProcessEvent		Identifies a process event	Yes	Param
	commandId	The command identifier.	Yes	-
	type	Application-specific string that identifies the type of event – it is recommended to use fully-qualified names (such as package names or URNs) in order to ensure uniqueness of event types.	Yes	-
Param		A parameter must have its name and value attributes specified. The only restriction that applies in the case of this element is that there must be a single parameter element with a given name per ProcessEvent.	Yes	-

Fault

A SOAP fault is sent back to the requesting process if an error occurs while processing the request. The fault message in this case is defined by the SOAP specification and does not any command (from Corus's protocol perspective); it therefore does not require a command identifier.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <SOAP-ENV:Fault>
            <faultcode>... some code ...</faultcode>
            <faultactor>... some actor ...</faultactor>
            <faultstring>... some message ...</faultstring>
            <detail>... some details ...</detail>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Element	*Description*	*Mandatory*	*Children*
Fault	Indicates that an error occurred while processing the request.	Yes	faultcode, faultactor, faultstring, detail
faultcode	An arbitrary error code.	Yes	-
faultactor	The actor that is responisble of the error	Yes	-
faultstring	An arbitrary error message.	Yes	-
detail	Details about the error – most likely a Java stack trace.	No	-