Implementing a JAIN SLEE Resource Adaptor
A quick-starter’s guide
by Michael Maretzke
5th October 2005

Introduction
“JAIN SLEE? JSLEE? A resource adaptor? What’s that all about? I’ve got a network protocol stack written in what-so-ever language and need to integrate it into a JSLEE application server – how should I do this?” These are questions which are addressed in this paper.

What is JAIN SLEE?
To make it very short and simple: “JSLEE is a low latency and high throughput application server for event processing designed for stringent requirements of core network signalling applications providing a distributed component model and a standardized framework.”

Initially, JSLEE was designed for network signalling environments. However, the whole architecture proved to be generic enough to allow for other application areas as well.

What is Mobicents?
“Mobicents is an Open Source VoIP Platform. Mobicents is the First and Only Open Source JAIN SLEE 1.0 Certified product, which brings to telecom application developers what J2EE brings to Web and Enterprise application developers.

In the scope of telecom Next Generation Intelligent Networks (NGIN), Mobicents fits in as a high-performance core engine for Service Delivery Platforms (SDP) and IP Multimedia Subsystems (IMS).”

For the scope of this paper and the idea of creating a simple-to-use and easy-to-understand Resource Adaptor which helps to increase the understanding of JSLEE technology in the industry the Mobicents application server was selected for the Resource Adaptor. Mobicents can be obtained freely at www.mobicents.org and is quite stable and very simple to install and work with. On the website a lot of information can be found on how to install and compile the application server.

What is a Resource Adaptor?
JSLEE is an application server, including an application model, which is based on components – the so called Service Building Blocks (Sbb). The whole application model is agnostic to whatever networking protocol or event source is utilized to trigger the execution of the coded business logic. Events in the JSLEE application model are POJOs (Plain Old Java Objects) and need to be created somewhere. The Resource Adaptor (RA) in JSLEE bridges the application model and the underlying event infrastructure. The event source could be everything emitting events implemented in whatever language and environment. Examples for event sources are a SIP stack, a JCC stack, a TCP/IP stack or even a HTTP stack. The RA accepts arriving protocol signals or specific events, creates the Java representations and fires them into the JSLEE application server.

The Structure of a RA
A JSLEE RA consists basically of a Resource Adaptor Type and a Resource Adaptor. The RA Type specifies the Events emitted by this class of RAs, the shareable state information between application logic Sbbs and RA – the Activity Context, and the interface utilized by JSLEE application logic Sbbs to access RA functions. Usually, RA Types are defined by an industry with same interests. In telecommunication industries a Call Control RA Type or a SIP RA Type may be good examples. The RA implements exactly one RA Type at a time. JNI technology is used to integrate non-Java stacks. Usually, RAs are stateful and model an internal state machine of the protocol activities. The RA decides on incoming signals to alter the internal state and to notify the JSLEE. To allow Sbbs to access the state of a RA, Activities and Activity Contexts (AC) are introduced.

An Activity, for example, represents one phone call or a game session. Incoming signals are mapped to one session-unique AC (the interface to the Activity) via the Activity Handle. The AC is accessible both from the RA and the Sbbs.
To conclude, the RA Type defines the kind of RA, the RA implementation wraps a specific protocol stack end emits Java objects as Events into the JSLEE application server and the AC is established to exchange state information between the RA and the Sbbs.

Confused? Don’t worry; let’s get down to the implementation!

The RAFrame Example

What does this example demonstrate? The Resource Adapter Framework (RAFrame) example shows how an existing Java-based protocol stack (RAFStack) is integrated into a JSLEE environment. Therefore, the RAFrame Resource Adaptor and all needed interfaces and classes are discussed in greater detail. The dependencies between Java code and deployment descriptors are illustrated and a simple example service (BounceSbb) is explained.

The source code for the example can be downloaded from the maretzke.com web site.

The RA splits in 6 packages:
- com.maretzke.raframe.message
- com.maretzke.raframe.ra
- com.maretzke.raframe.ratype
- com.maretzke.raframe.stack
- com.maretzke.raframe.test.server
- com.maretzke.raframe.test.client

The service’s package is:
- com.maretzke.raframe.service.bounce

Both, the RA and the service ship with build.xml files for automated building and packaging. The process of building and deploying the RA is explained later.

The Protocol

For the purpose of creating an example RA for mainly (self-)educational purposes, a simple and easy-to-understand protocol was needed. The RAFrame protocol was born. It is TCP/IP based and follows this format rules:

<table>
<thead>
<tr>
<th>ID</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>INIT</td>
</tr>
</tbody>
</table>

Protocol messages may contain the commands INIT, ANY and END.

A session is started with the INIT command. Next any number of ANY commands may occur. The session terminates with the END command. Any other sequence of commands is considered to be invalid.

The Protocol Stack

The stack for the described protocol consists of three classes and can be located in the package com.maretzke.raframe.stack.

The class RAFStack contains the stack logic and implements a TCP/IP ServerSocket. Furthermore, it offers logic to send information to another RAFStack implementation.

An incoming TCP/IP connection let the RAFStack instantiate a new RASTackThread and delegates the work for the incoming information. The RASTackThread reads the incoming information and informs listening instances. These objects implement the
RAFStackListener interface and have previously registered for notification. The stack implementation is multi-threaded to decrease idle times needed for socket communication.

**Testing the Protocol Stack**

To see the protocol stack working, open two command line windows (cmd.exe), change to the RAFrame\bin directory and execute startRAFServer.bat in the first window and the Swing version of the client with startSwingRAFClient.bat in the second window.

![Figure 3 The RAFSwingClient](image)

The server and the client both utilize the RAFStack classes to communicate. The according classes are located in the package com.maretzke.raframe.test.client and com.maretzke.raframe.test.server. Now, we have a stack implementation which allows TCP/IP based communication between server and client instances. However, the stack does not ensure that the defined protocol is not violated. That will be one of the tasks for the RA.

**Deployment Descriptors**

Well known (and hated) from the J2EE world, JSLEE applies the concepts of Deployment Descriptors (DD) for configuration, deployment and packaging. The various DDS and their meaning for the different JSLEE entities are explained later on despite one: deployable-unit.xml. It is important for packaging purposes and references all elements contained in a JSLEE archive ready for deployment – the deployable unit.

![Figure 4 Overview of various Deployment Descriptors](image)

**The RAFrame Events**

As mentioned before, Events are means to communicate between RAs and Sbbs in the context of JSLEE. In the case of RAFrame RA the encoding of the protocol messages in Java objects is quite simple.

```java
public interface Message {
    public final static int INIT = 1;
    public final static int ANY = 2;
    public final static int END = 3;
    public String getId();
    public String getCommand();
    public int getCommandId();
}
```

*Listing 1 Interface Message*

The Java interface Message abstracts a protocol message of the RAFStack. The concrete Message object is wrapped in a MessageEvent object. Events, ready! Objects implementing this interface are created by the RA and delivered into the JSLEE environment.

```java
public interface MessageEvent {
    public Message getMessage();
    public Object getSource();
}
```

*Listing 2 Interface MessageEvent*

To create real objects of Message and MessageEvent the RA and Sbbs both utilize a MessageFactory object.

```java
public interface MessageFactory {
    public Message createMessage(String id, String command);
    public MessageEvent createMessageEvent(Object obj, Message message);
}
```

*Listing 3 Interface MessageFactory*

All mentioned classes are located in the package com.maretzke.raframe.message.

**The RA Type RAFrame**

The definition of the RA Type’s Events happens in the DD event-jar.xml. The tag event-type-name defines a unique name for Events handled by the RA Type. This name is mapped
Implementing a JAIN SLEE Resource Adaptor

Listing 4 Excerpt of event-jar.xml

The RA Type’s Activity and Activity Context

The DD resource-adaptor-type-jar.xml defines amongst others the Activity and Activity Context for the RA Type.

Listing 5 Activity definition in resource-adaptor-type-jar.xml

The tag activity-type-name refers to the Java interface RAFActivity. Its implementation represents the shareable state for one single Activity between the RA and the Sbb.

Listing 6 Interface RAFActivity

Remember, an Activity is the representation of a state, specific for exactly one sequence of Events, for example the establishment of a call or the setup of an online game session. One specific Activity is identified by a unique Activity Handle: here the incoming message’s identifier. In our example the class RAFActivityHandle implements the Handle. Listing 7 shows the generation and lookup of Activities.

Listing 7 Excerpt of RAFrameResourceAdaptor.onEvent()

On the other hand, Sbbs can access the Activity through the ActivityContextInterface as shown in Listing 8.

Listing 8 Excerpt of BounceSbb’s onInitEvent() method

The RA Type’s offering for Sbbs

The DD defines also the interface between Sbb components and the RA. This interface is the only way for Sbb components to interact with the RA.

Listing 9 RA interface facing Sbbs definition in resource-adaptor-type-jar.xml

The class com.maretzke.raframe.ra.RAFrameProviderImpl implements the interface. Sbbs may ask for a MessageFactory object to create Message and MessageEvent objects and access the send() method of the RA implementation for sending Message objects.

How does this work?

Listing 10 Interface RAFrameResourceAdaptorSbbInterface

The class com.maretzke.raframe.ra.RAFrameProviderImpl implements the interface. Sbbs may ask for a MessageFactory object to create Message and MessageEvent objects and access the send() method of the RA implementation for sending Message objects.

How does this work?

Listing 11 Excerpt of BounceSbb’s onAnyEvent() method

The BounceSbb needs to query the JNDI tree for the Sbb’s interface to the RA. The JNDI name is configured in the DD of the Sbb.

to a concrete Java class specified by the tag event-class-name.

Listing 3 Excerpt of event-jar.xml
The RA – Structure and Methods

The class RAFrameResourceAdaptor implements the RAFrame RA and can be located in the package com.maretzke.raframe.ra. Most of the RA’s methods implement the interface javax.slee.ResourceAdaptor and represent hooks invoked by the JSLEE environment during the lifecycle of the RA. In JSLEE version 1.0 (JSR-22) the Resource Adaptor integration was considered an implementation specific detail. JSLEE version 1.1 (JSR-240) focuses on the Resource Adaptor architecture, however is at the time being not finished. The open source project Mobicents follows the standard activities very closely and implements already most of the aspects of JSLEE version 1.1. So, if you encounter differences or abnormalities they could be motivated by following the standard quite closely.

entityCreated() and entityActivated()

The method entityCreated() is the very first method called from the JSLEE environment after the instantiation of the RA.

```java
public void entityCreated(BootstrapContext bootstrapContext)
    throws ResourceException {
    this.bootstrapContext = bootstrapContext;
    this.sleeEndpoint = bootstrapContext.getSleeEndpoint();
    this.eventLookup = bootstrapContext.getEventLookupFacility();
    stack = null;
}
```

Listing 12 the method entityCreated()

The method initializes important references to JSLEE objects provided by the BootstrapContext object. Afterwards JSLEE invokes entityActivated() to allow the RA to finalize initialization work to be done.

```java
public void entityActivated()
    throws ResourceException {
    messageFactory = new MessageFactoryImpl();
    raProvider = new RAFrameProviderImpl(this, messageFactory);
    messageParser = new RAFMessageParser();
    stack = new RAFStack(port, remotehost, remoteport);
    stack.addListener(this);
    stack.start();
    initializeNamingContext();
    activities = new HashMap();
}
```

Listing 13 the method entityActivated()

In the method, the RA creates the RAFStack object, registers itself as listener and starts the stack. At the end, a new HashMap object is created to store activities.

**onEvent()**

The embedded RAFStack object of the RA invokes onEvent() and hands over the received characters as an argument.

In onEvent() the incoming information is parsed and discarded if invalid.

```java
public void onEvent(String incomingData) {
    MessageEvent event;
    Address address;
    int eventID;
    // parse the incoming data
    try {
        Message message = messageParser.parse(incomingData);
        event = messageFactory.createMessageEvent(this, message);
    }
    catch (IncorrectRequestFormatException irfe) {
        // Unfortunately, the incoming messsage does not comply
        // with the protocol / message
        // format rules. The message is discarded.
        return;
    }
    // generate the activity handle which uniquely identifies
    // the appropriate activity context
    RAFActivityHandle handle = new RAFActivityHandle
        (event.getMessage().getId());
    // lookup the activity
    RAFActivity activity = (RAFActivity)
        activities.get(handle);
    // activity does not exist - let's create one
    if (activity == null) {
        activity = new RAFActivityImpl();
        activities.put(handle, activity);
    }
    The validity of the incoming message according
    to the protocol rules is checked utilizing the
    Activity’s state machine (isValid()).

    if (!activity.isValid(event.getMessage().getCommandId())) {
        // Not a valid command. Command corrupts rules defined for
        // the protocol
        return;
    }
    The identifier of the Event is looked up in the
    JNDI tree. If not found in the JNDI tree, it is
    assumed a non-known and therefore invalid
    message.

    // the fireEvent() method needs a default address to where
    // the events should be fired to
    address = new Address(AddressPlan.IP, "127.0.0.1");
    // get the eventID from the JNDI tree
    try {
        eventID = eventLookup.getEventID(this, "maretzke", "1.0");
    }
    catch (…) {
        return;
    }
    finally, the message is handed over to the
    JSLEE environment. This is done via the
    SleeEndpoint. Is the message an END
    message, the JSLEE environment is notified
    with activityEnding() otherwise with
    fireEvent().

    try {
        if (event.getMessage().getCommand().toLowerCase().
            compareTo("end") == 0) {
            // activityEnding()
            return;
        }
    }
```

Version 1.1 -5 - 5th October 2005
// if the command is an end command, the connected
// activity needs to end
// this is signalled to the SLEE via activityEnding()
seeEndpoint.activityEnding(
    new RAFActivityHandle(event.getMessage().getId()));

else {
// fire the event into the SLEE and proceed
seeEndpoint.fireEvent(
    new RAFActivityHandle(event.getMessage().getId()),
    (Object) event, eventID, address);
}
} catch (...) {
...
}
}

Listing 14 the method onEvent()

getActivity() and ActivityEnded()
The method getActivity() returns the Activity object associated with one specific
Handle.

public Object getActivity(ActivityHandle activityHandle) {
    return activities.get(activityHandle);
}

Listing 15 the method getActivity()

When a specific Activity ends, the RA is notified
by the JSLEE environment via the invocation of
the activityEnded() method. Here, the RA
removes the Activity identified by its Handle
from the HashMap object.

public void activityEnded(ActivityHandle activityHandle) {
    activities.remove(activityHandle);
}

Listing 16 the method activityEnded()

To summarize the lifecycle of Activity objects:
One specific Activity is created in the RA’s
onEvent() method when an initial Event is
received – here: the INIT message. During the
lifetime of the Activity it is accessed via the
getActivity() method of the RA. As soon as
the Activity ends, the JSLEE environment
notifies the RA by invoking activityEnded(). Now the RA removes
the Activity from its storage.

RA Deployment Descriptor

The DD resource-adaptor-jar.xml defines
a name for the RA and associates the RA with a
RA Type. Furthermore, the RA’s class is
specified.

<resource-adaptor-type-ref>
    <resource-adaptor-type-name>
        rafframe
    </resource-adaptor-type-name>
    <resource-adaptor-type-vendor>
        maretzke
    </resource-adaptor-type-vendor>
    <resource-adaptor-type-version>1.0</resource-adaptor-type-version>
</resource-adaptor-type-ref>

The RA Type for our RA specifies three Events:
INIT, ANY and END. The tag event-type-
name refers to the event-definition found
in the DD event-jar.xml (Listing 4).

Building the RA

Building the RA is quite simple. Change to the
RAFrame directory and run ant. As a
prerequisite, ant (e.g. ant 1.6.1) and Java (e.g.
Java 1.5.0_04) needs to be installed properly.
For an overview of valid ant targets, type ant
help.

After running ant the dist directory should be
populated with three files:

- rafframe-1.0.jar
- rafframe-local-ra.jar
- rafframe-ra-type.jar

The first one contains the build directory and
will be used during service compilation. The
latter both are the RA and the RA Type
deployable unit archives. They can be deployed
into a running Mobicents JSLEE
implementation.

Deploying the RA

Deploying the RA could be done two different
ways: one easy and automated way and the hard
and manual way.
Common for both approaches are the steps to
install the RA. First, the RA Type archive needs
to be installed. Next, the RA archive is installed.
The other way around will fail because the RA
relies on the RA Type. Next, the RA entity needs
to be created. In this step, an association between
the RA ‘rafframe#maretzke#1.0’ as described in
the DD (see Listing 17) and the entity name
‘RAFrameRA’ will be created. Afterwards, the
entity will be activated and in a last step an
entity link will be created.
The manual way
After starting the Mobicents JSLEE point a WWW browser to http://localhost:8080/jmx-console/. Type into the text field slee:* to filter the view. Click on name=DeploymentMBean. Scroll down and look for the method install with one String parameter. Enter the URL where the file raframe-ra-type.jar is located (e.g.: file:///D:/RAFrame/dist/raframe-ra-type.jar) in the text field and press the invoke button. The archive will be installed and a deployable unit identifier is returned (e.g. DeployableUnitID[0]).
Install the RA by repeating the steps above with the file raframe-local-ra.jar. The deployable unit identifier for the RA is returned (e.g. DeployableUnitID[1]).
Return to the JMX Agent View and click on name=ResourceAdaptorMBean. Scroll down and look for the method createResourceAdaptorEntity which accepts three parameters. The first parameter is of type ResourceAdaptorID. Enter ResourceAdaptorID[raframe#maretzke#1.0] into the p1 text field and RAFrameRA into the p2 text field. The p3 text field remains empty. Press the invoke button. The result page says “Operation completed successfully without a return value."
Return to the MBean view and look for the activateResourceAdaptorEntity method. In the text field for p1 enter RAFrameRA. Press the invoke button. Again, the result page says “Operation completed successfully without a return value."
As the final step, return to the MBean view and look for the createEntityLink method. Enter RAFrameRA in both text fields, p1 and p2. Press the invoke button. Again, the result page says “Operation completed successfully without a return value."
Congratulations, you’ve just deployed the RAFrame RA successfully!
To see what happened in the background have a look on the console window of the Mobicents application server.

The automated way
Obviously, the manual way is fairly complex and time consuming. Fortunately, there is an automated way supported by a script stored in the bin directory.

Change to the bin folder in the RAFrame directory and execute DeployRAFRA.bat. Finished!
Switching to the console window of Mobicents shows exactly the same output as in the case of manual deployment.

BounceSbb – utilizing the RA
Until now, we concentrated on the RA, on extending and customizing the JSLEE environment. From now on, we focus on using the newly created RA by an application.
The BounceSbb service is pretty simple. Incoming messages of type INIT and END increase the Activity’s counter for each of the commands. The ANY command Event handler does the same and utilizes the RA to send the command string back to the sender adding the prefix “Command bounced by BounceSbb: “. That’s it!

The Sbb’s Methods
The source for the class BounceSbb is located in the package com.maretzke.raframe.service.bounce. The Sbb implements the interface javax.slee.Sbb. All methods starting with sbb… or containing Sbb in their names are inherited from this interface. Most of them are called by the JSLEE environment according to the Sbb’s lifecycle defined in the JSLEE specification. For BounceSbb, the only method of value is setSbbContext().

setSbbContext()

Listing 18 Excerpt of the method setSbbContext()
The method queries the JNDI tree for the trace facility provided by the JSLEE environment and for the interface to the RA. This interface is used in the Event handler method for the ANY command (see Listing 11) to send the answer back to the initial sender.

Event handling
The most important methods in BounceSbb are the Event handler onInitEvent(), onAnyEvent
and onEndEvent(). BounceSbb declares in its DD sbb-jar.xml (see Listing 20 below) the interest in receiving INIT, ANY and END Events. Referring to the tag event-name in the DD, the Sbb has to implement on<event-name>() methods. Listing 19 below shows the event handler for ANY events.

When invoking the method the JSLEE environment hands over a MessageEvent object and the ActivityContextInterface.

```java
public void onAnyEvent(MessageEvent event, ActivityContextInterface ac) {
    trace(Level.INFO, "BounceSbb: " + this + " received event: " + event.getMessage().getId() + " Command = " + event.getMessage().getCommand());
}
```

The ActivityContextInterface object is used to access the Activity shared by the Sbb and the RA.

BounceSbb alters the Activity object.

```java
RAFActivity activity = RAFActivity ac.getActivity();
activity.anyReceived();
```

Listing 19 BounceSbb’s onAnyEvent() method

**Sbb Deployment Descriptor**

The DD sbb-jar.xml for the service defines the representing Java class for the service, lists all the Events the Sbb wants to receive, binds the Sbb to a specific RA Type and names the JNDI bindings to one specific RA instance deployed in the Sbb environment.

Listing 20 Excerpt sbb-jar.xml

**Building the Sbb**

Copy the file raframe-1.0.jar from the folder RAFrame\dist to the folder RASbb\lib. Then change to the RASbb directory and execute ant. As said previously, ant and Java needs to be installed properly. And again, ant help lists the valid targets for this project.

After executing ant, the dist directory should contain the archive bouncesbb-service.jar. The archive represents a deployable unit for JSLEE and could be dropped into a running Mobicents JSLEE environment – with previously installed RAFrame RA!

**Deploying the Sbb**

The manual way

As described for the RA, navigate to the DeploymentMBean and search for the install method. Enter the URL to the file bouncesbb-service.jar in the text field and invoke the method. The operation delivers something like DeployableUnitID[2]. Now, the service is installed. Next step is to activate the service.

Navigate to the ServiceManagementMBean and look for the activate method that accepts a javax.slee.ServiceID parameter. Enter
Implementing a JAIN SLEE Resource Adaptor

the value ServiceID[Resource Adaptor Framework Bounce Service#maretzke#1.0] in the text field and invoke the method. The service is deployed and waiting for Events.

The automated way

Again, a script will help speed up deployment. Change to the bin directory located in the Sbb folder. Execute DeployBounceSbb.bat and finished!

Testing the installation

And now? Exciting things happened, however most of them are not visible …

After installing the RAFrame RA and the BounceSbb as described above, the Mobicents JSLEE environment is set up to receive RAFStack protocol messages as defined earlier.

Let’s start experimenting. Change to RAFrame\bin and execute startSwingRAFClient.bat.

The RAFrame RA is started and the RAFFrame RA fires event into SLEE. EventID: 13; CallID: 102; Command: INIT

The RAFStack inside the RA received the characters.

The RA gets notified, looks for the Activity Handle and delivers the Event to the JSLEE.

The JSLEE environment creates a BounceSBB object and invokes its lifecycle methods.

The Sbb is up and deals with the incoming Event.

Understand what happens

Great, let’s follow the characters typed into the swing client. Pressing the “Send” button invokes the method shown in Listing 21.

private void sendBtnActionPerformed(ActionEvent evt) {
    stack.send(inputField.getText());
    outputArea.setText(outputArea.getText() + "Send -----> "+inputField.getText() + '\n');
}

Listing 21 sendBtnActionPerformed in RAFSwingClient
The next segment shows processing of the END Event.

```
[RAFStackThread] Serverthread Thread-78 started.
[RAFStackThread] bytes received (7) = 102 END
[RAFFrameResourceAdaptor] Incoming request: 102 END
[RAFActivityHandle] RAFrameResourceAdaptor.onEvent():
  RAFrameRA fires event into SLEE. EventID: 14; CallID: 102; Command: ANY
  RAFrameRA.firesEventIntoSLEE() called.
[RAFFrameResourceAdaptor] RAFrameResourceAdaptor.onEvent():
  RAFrameRA.firesEventIntoSLEE() called.
[RAFFrameResourceAdaptor] RAFrameResourceAdaptor.onEvent():
  RAFrameRA.firesEventIntoSLEE() called.
```

The last snippet shows the rejection of the ...

```
Listing 23 Console output of the Mobicents JSLEE for "102 ANY, 102 END, 102 ANY"

What’s next?

RAFrame is already quite a complex construct from a programming perspective and if you kick-start into SLEE programming it models. However from a communication and protocol perspective RAFrame represents quite a simple protocol and interaction model. If time allows, the example needs more attention on transactions, a more complex state model, a more demanding protocol model and some interaction initiated by Sbb's. For questions and comments please contact me via michael@maretzke.com.

```