JBoss Communications JAIN SLEE Sip Service Example User Guide

by Eduardo Martins, Bartosz Baranowski, and Alexandre Mendonça

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts* [https://fedorahosted.org/liberation-fonts/] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight key caps and key-combinations. For example:

> To see the contents of the file `my_next_bestselling_novel` in your current working directory, enter the `cat my_next_bestselling_novel` command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a key cap, all presented in Mono-spaced Bold and all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the hyphen connecting each part of a key-combination. For example:

> Press **Enter** to execute the command.
>
> Press **Ctrl**+**Alt**+**F1** to switch to the first virtual terminal. Press **Ctrl**+**Alt**+**F7** to return to your X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in `Mono-spaced Bold`. For example:

> File-related classes include `filesystem` for file systems, `file` for files, and `dir` for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialogue box text; labelled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System > Preferences > Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications > Accessories > Character Map** from the main menu bar. Next, choose **Search > Find** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit > Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in Proportional Bold and all distinguishable by context.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

`Mono-spaced Bold Italic` or ***Proportional Bold Italic***

Whether Mono-spaced Bold or Proportional Bold, the addition of Italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type `ssh` *`username@domain.name`* at a shell prompt. If the remote machine is `example.com` and your username on that machine is john, type `ssh john@example.com`.
>
> The `mount -o remount` *`file-system`* command remounts the named file system. For example, to remount the `/home` file system, the command is `mount -o remount /home`.
>
> To see the version of a currently installed package, use the `rpm -q` *`package`* command. It will return a result as follows: *`package-version-release`*.

Note the words in bold italics above username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as

a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules* (*MPMs*). Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

1.2. Pull-quote Conventions

Two, commonly multi-line, data types are set off visually from the surrounding text.

Output sent to a terminal is set in `Mono-spaced Roman` and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in `Mono-spaced Roman` but are presented and highlighted as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
  public static void main(String args[])
      throws Exception
  {
    InitialContext iniCtx = new InitialContext();
    Object         ref    = iniCtx.lookup("EchoBean");
    EchoHome       home   = (EchoHome) ref;
    Echo           echo   = home.create();

    System.out.println("Created Echo");

    System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
  }

}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

A note is a tip or shortcut or alternative approach to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring Important boxes won't cause data loss but may cause irritation and frustration.

Warning

A Warning should not be ignored. Ignoring warnings will most likely cause data loss.

2. Provide feedback to the authors!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in the the *Issue Tracker* [http://bugzilla.redhat.com/bugzilla/], against the product **JBoss Communications JAIN SLEE Sip Service Example**, or contact the authors.

When submitting a bug report, be sure to mention the manual's identifier: JAIN_SLEE_SipServices_EXAMPLE_User_Guide

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Introduction to JBoss Communications JAIN SLEE Sip Service Example

This example is a JAIN SLEE application which processes SIP Messages from SIP UAs to act as a simple proxy and registrar.

Example implements proxy routines defined in *section 16* [http://tools.ietf.org/html/rfc3261#section-16] and registrar routines defined in *section 10* [http://tools.ietf.org/html/rfc3261#section-10] of RFC3261.

It is not a trivial example as it provides usage of

- child relations
- timers
- null activities
- SBB activity context interfaces
- activity context interfaces variables
- SBB environment entries
- SBB local interfaces
- SBB local interfaces
- JAIN SIP RA code

Thus it should be considered as target for more advanced users.

Example service is capable of routing SIP messages based on registrar entries or target domain. Registrar entries are based on received SIP REGISTER request for example local domains.

Setup

2.1. Pre-Install Requirements and Prerequisites

Ensure that the following requirements have been met before continuing with the install.

2.1.1. Hardware Requirements

The Example doesn't change the JBoss Communications JAIN SLEE Hardware Requirements, refer to JBoss Communications JAIN SLEE documentation for more information.

2.1.2. Software Prerequisites

The Example requires JBoss Communications JAIN SLEE properly set, with SIP11 Resource Adaptor deployed.

2.2. JBoss Communications JAIN SLEE Sip Service Example Source Code

This section provides instructions on how to obtain and build the Sip Service Example from source code.

2.2.1. Release Source Code Building

1. **Downloading the source code**

Important

Subversion is used to manage its source code. Instructions for using Subversion, including install, can be found at *http://svnbook.red-bean.com*

Use SVN to checkout a specific release source, the base URL is ?, then add the specific release version, lets consider 2.3.0.FINAL.

```
[usr]$ svn co ?/2.3.0.FINAL slee-example-sip-services-2.3.0.FINAL
```

2. **Building the source code**

Important

Maven 2.0.9 (or higher) is used to build the release. Instructions for using Maven2, including install, can be found at *http://maven.apache.org*

Use Maven to build the deployable unit binary.

```
[usr]$ cd slee-example-sip-services-2.3.0.FINAL
[usr]$ mvn install
```

Once the process finishes you should have the `deployable-unit` jar file in the `target` directory, if JBoss Communications JAIN SLEE is installed and environment variable JBOSS_HOME is pointing to its underlying JBoss Enterprise Application Platform directory, then the deployable unit jar will also be deployed in the container.

2.2.2. Development Trunk Source Building

Similar process as for *Section 2.2.1, "Release Source Code Building"*, the only change is the SVN source code URL, which is NOT AVAILABLE.

2.3. Installing JBoss Communications JAIN SLEE Sip Service Example

To install the Example simply execute provided ant script `build.xml` default target:

```
[usr]$ ant
```

The script will copy the Example's deployable unit jar to the `default` JBoss Communications JAIN SLEE server profile deploy directory, to deploy to another server profile use the argument `-Dnode=`.

Note

Some services may wish to receive `INVITE` request. Call:

```
[usr]$ ant deploy-all-without-initial-invite
```

to deploy Sip Service configuration which does not react to `INVITE`

2.4. Uninstalling JBoss Communications JAIN SLEE Sip Service Example

To uninstall the Example simply execute provided ant script `build.xml` `undeploy` target:

```
[usr]$ ant undeploy
```

The script will delete the Example's deployable unit jar from the `default` JBoss Communications JAIN SLEE server profile deploy directory, to undeploy from another server profile use the argument `-Dnode=`.

Note

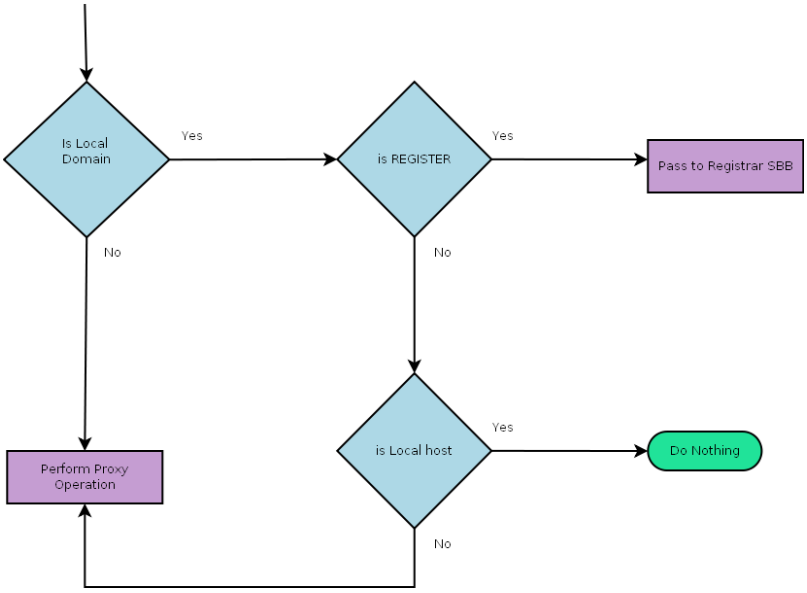
Some services may wish to receive `INVITE` request. Call:

```
[usr]$ ant undeploy-all-without-initial-invite
```

to undeploy Sip Service configuration which does not react to `INVITE`

Design Overview

The Sip Service Example is JAIN SLEE 1.1 Application which handles incoming SIP messages. Depending on message target and message method, example takes different action. General rule of message traversal through this example look as follows:



Sip Service Example Flow

3.1. Example Components

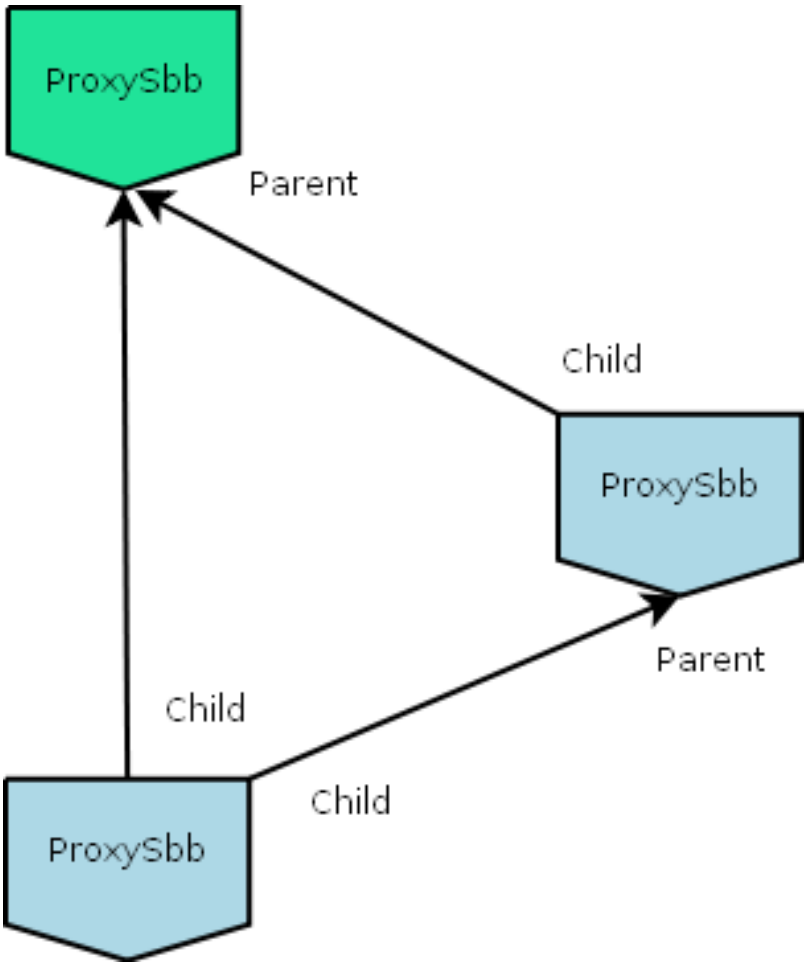
Example consist of three SBBs. Each one performs different task:

ProxySbb
: Is responsible for routing procedures. It performs operations defined in *section 16* [http://tools.ietf.org/html/rfc3261#section-16] of RFC3261 and sends message to proper node. It is first SBB to receive any incoming message.

RegistrarSbb
: Is responsible for proper REGISTER request processing. It performs operations defined in *section 10* [http://tools.ietf.org/html/rfc3261#section-10] of RFC3261. Registrar creates `AOR(Address Of Record)` and based on REGISTER content performs update to location data.

LocationSbb
: Is responsible for storing and invalidating entries for AORs. That is - it stores mapping between AOR and contact addresses, and on contact address expiration, removes it.

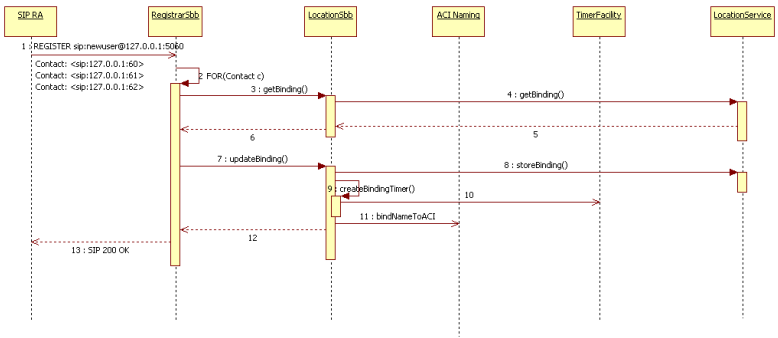
Root SBB of service is ProxySbb, which receives messages as first. Relation between SBB s look as follows:



Sip Service Example components relations

3.2. Location and Registrar Operations

REGISTER requests which are passed to RegistrarSbb trigger routine which updates user contact database and associated expiration date. Simple sequence diagram for success case look as follows:



Sip Service Registrar and Location sequence

Location SBB creates JSLEE timer for each pair AOR-contact address. Once timer expires contact address is considered as invalid. Time between creation of this mapping and expiration depends on `Expires Header` content.

Source Code Overview

Important

To obtain the example's complete source code please refer to *Section 2.2, "JBoss Communications JAIN SLEE Sip Service Example Source Code"*.

Chapter *Chapter 3, Design Overview* explains top level view of example. This chapter explains how components perform their tasks. For more detailed explanation of JSLEE related source code and xml descriptors, please refer to simpler examples, like `sip-wakeup`

4.1. ProxySbb overview

Proxy SBB is top component in this example - it is declared as root SBB . It declares event handlers for all SIP11 ResourceAdaptor transactional events.

ProxySbb expects transactional events to be fired, that is, it assumes that if in-Dialog event is fired, some other application is responsible for messages. `ProxySbb` expects also that at any given time it is attached only to one incoming transaction(Server) and one or more ongoing(Client). That is because ProxySbb uses `SbbContext` facility to retrieve list of current activities it is attached to. Class `org.mobicents.slee.services.sip.proxy.ProxySbb` has all the logic linking JSLEE with routing logic declared in inner class `org.mobicents.slee.services.sip.proxy.ProxySbb $ProxyMachine` .

4.1.1. Initial event handler

As root of service proxy declares requests as initial. It does that with xml descriptor of event handler in sbb-jar.xml, for instance:

```
<event event-direction="Receive" initial-event="True">
   <event-name>InviteEvent</event-name>
   <event-type-ref>
      <event-type-name>javax.sip.message.Request.INVITE</event-type-name>
      <event-type-vendor>net.java.slee</event-type-vendor>
      <event-type-version>1.2</event-type-version>
   </event-type-ref>
   <initial-event-selector-method-name>
      callIDSelect
   </initial-event-selector-method-name>
</event>
```

Event handlers declaration has `initial-event-selector-method-name` element. This element identifies callback method name, which is invoked by `JSLEE` for all fired events. This callback is used by `JSLEE` to determine if event should create `SBB Entity` if it does not exists and determine name which distinguishes it. `ProxySbb` has following definition of initial event handler callback:

```
public InitialEventSelector callIDSelect(InitialEventSelector ies) {
    Object event = ies.getEvent();
    String callId = null;
    if (event instanceof ResponseEvent) {
        ies.setInitialEvent(false);
        return ies;
    } else if (event instanceof RequestEvent) {
        // If request event, the convergence name to callId
        Request request = ((RequestEvent) event).getRequest();
        if (!request.getMethod().equals(Request.ACK)) {
            callId = ((ViaHeader) request.getHeaders(ViaHeader.NAME).next())
                .getBranch();
        } else {
            callId = ((ViaHeader) request.getHeaders(ViaHeader.NAME).next())
                .getBranch()
                    + "_ACK";
        }
    }
    // Set the convergence name
    if (logger.isDebugEnabled()) {
        logger.debug( "Setting convergence name to: " + callId);
    }
    ies.setCustomName(callId);

    return ies;
}
```

Response events are not considered as initial because there should be SBB Entity attached to client transactions `activity context interface. This entity will handle those responses.`

4.1.2. Request handlers

Request handlers are only responsible for relying message to `ProxyMachine` . Example request handler look as follows:

```
public void onInviteEvent(RequestEvent event, ActivityContextInterface ac) {

  if (logger.isDebugEnabled())
    logger.debug("Received INVITE request");

  processRequest(event.getServerTransaction(), event.getRequest(), ac);
}

private void processRequest(ServerTransaction serverTransaction,
    Request request, ActivityContextInterface ac) {

if (logger.isInfoEnabled())
    logger.info("processing request: method = \n"
            + request.getMethod().toString());


try {

    if (getServerTransactionTerminated()) {
        if (logger.isDebugEnabled())
            logger.debug("[PROXY MACHINE] txTERM \n" + request);
        return;
    }

    // if (getServerTX() == null)
    // setServerTX(serverTransaction);
    // Go - if it is invite here, serverTransaction can be CANCEL
    // transaction!!!! so we dont want to overwrite it above
    new ProxyMachine(getProxyConfigurator(), getLocationSbb(),
            this.addressFactory, this.headerFactory,
            this.messageFactory, this.provider)
        .processRequest(serverTransaction, request);

} catch (Exception e) {
    // Send error response so client can deal with it
    logger.warn( "Exception during processRequest", e);
    try {
```

```
            serverTransaction.sendResponse(messageFactory.createResponse(
                    Response.SERVER_INTERNAL_ERROR, request));
        } catch (Exception ex) {
            logger.warn( "Exception during processRequest", e);
        }
    }

}
```

`ProxyMachine` class performs all required operations to forward request :

- check if request is properly built.
- process information in request and determine targets.
- forward message to all possible targets, attach to outgoing legs.

```
class ProxyMachine extends MessageUtils implements MessageHandlerInterface {
protected final Logger log = Logger.getLogger("ProxyMachine.class");

// We can use those variables from top level class, but let us have our
// own.

protected LocationService reg = null;

protected AddressFactory af = null;

protected HeaderFactory hf = null;

protected MessageFactory mf = null;

protected SipProvider provider = null;

protected HashSet<URI> localMachineInterfaces = new HashSet<URI>();

protected ProxyConfiguration config = null;

public ProxyMachine(ProxyConfiguration config,
        LocationService registrarAccess, AddressFactory af,
        HeaderFactory hf, MessageFactory mf, SipProvider prov)
        throws ParseException {
```

```
        super(config);
        reg = registrarAccess;
        this.mf = mf;
        this.af = af;
        this.hf = hf;
        this.provider = prov;
        this.config = config;
        SipUri localMachineURI = new SipUri();
        localMachineURI.setHost(this.config.getSipHostname());
        localMachineURI.setPort(this.config.getSipPort());
        this.localMachineInterfaces.add(localMachineURI);
    }

    public void processRequest(ServerTransaction stx, Request req) {
        if (log.isDebugEnabled()) {
            log.debug("processRequest");
        }
        try {
            Request tmpNewRequest = (Request) req.clone();

            // 16.3 Request Validation
            validateRequest(stx, tmpNewRequest);

            // 16.4 Route Information Preprocessing
            routePreProcess(tmpNewRequest);

            // logger.debug("Server transaction " + stx);
            // 16.5 Determining Request Targets
            List targets = determineRequestTargets(tmpNewRequest);

            Iterator it = targets.iterator();
            while (it.hasNext()) {
                Request newRequest = (Request) tmpNewRequest.clone();
                URI target = (URI) it.next();

                // Part of loop detection, here we will stop initial reqeust
                // that makes loop in local stack
                if (isLocalMachine(target)) {

                    continue;
                }

                // 16.6 Request Forwarding
                // 1. Copy request
```

```
            // 2. Request-URI
            if (target.isSipURI() && !((SipUri) target).hasLrParam())
                newRequest.setRequestURI(target);

            // *NEW* CANCEL processing
            // CANCELs are hop-by-hop, so here must remove any existing
            // Via
            // headers,
            // Record-Route headers. We insert Via header below so we
            // will
            // get response.
            if (newRequest.getMethod().equals(Request.CANCEL)) {
                newRequest.removeHeader(ViaHeader.NAME);
                newRequest.removeHeader(RecordRouteHeader.NAME);
            } else {
                // 3. Max-Forwards
                decrementMaxForwards(newRequest);
                // 4. Record-Route
                addRecordRouteHeader(newRequest);
            }

            // 5. Add Additional Header Fields
            // TBD
            // 6. Postprocess routing information
            // TBD
            // 7. Determine Next-Hop Address, Port and Transport
            // TBD

            // 8. Add a Via header field value
            addViaHeader(newRequest);

            // 9. Add a Content-Leangth header field if necessary
            // TBD

            // 10. Forward Request

            ClientTransaction ctx = forwardRequest(stx, newRequest);

            // 11. Set timer C

        }

    } catch (SipSendErrorResponseException se) {
```

```
            se.printStackTrace();
            int statusCode = se.getStatusCode();
            sendErrorResponse(stx, req, statusCode);
        } catch (SipLoopDetectedException slde) {
            log.warn("Loop detected, droping message.");
            slde.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }


    ...


    }
```

4.1.3. Response handlers

Response handlers are invoked for incoming responses. In case multiple final responses incoming in sequence, first response is forwarded in stateful manner, others are forwarded in stateles manner. Example response handler look as follows:

```
    public void onRedirRespEvent(ResponseEvent event,
        ActivityContextInterface ac) {
    if (logger.isDebugEnabled())
        logger.debug("Received 3xx (REDIRECT) response");

    processResponse(event.getClientTransaction(), event.getResponse(), ac);
}

    private void processResponse(ClientTransaction clientTransaction,
        Response response, ActivityContextInterface ac) {


    if (logger.isInfoEnabled())
        logger.info("processing response: status = \n"
                + response.getStatusCode());

    try {
```

```
        if (getServerTransactionTerminated()) {
            return;
        }

        // Go
        ServerTransaction serverTransaction = getServerTransaction(clientTransaction);
        if (serverTransaction != null) {
            new ProxyMachine(getProxyConfigurator(), getLocationSbb(), this.addressFactory,
                    this.headerFactory, this.messageFactory, this.provider)
            .processResponse(serverTransaction,
                    clientTransaction, response);
        } else {
            logger.warn("Weird got null tx for[" + response + "]");
        }

    } catch (Exception e) {
        // Send error response so client can deal with it
        logger.warn( "Exception during processResponse", e);
    }

}
```

`ProxyMachine` class performs all required operations to forward response :

- check if response should be forwarded.
- forward it.

```
    public void processResponse(ServerTransaction stx,
            ClientTransaction ctx, Response resp) {

        // Now check if we really want to send it right away

        // log.info(this.getClass().getName(), "processResponse");

        try {

            Response newResponse = (Response) resp.clone();

            // 16.7 Response Processing
            // 1. Find appropriate response context
```

```
            // 2. Update timer C for provisional responses

            // 3. Remove topmost via
            Iterator viaHeaderIt = newResponse.getHeaders(ViaHeader.NAME);
            viaHeaderIt.next();
            viaHeaderIt.remove();
            if (!viaHeaderIt.hasNext())
               return; // response was meant for this proxy

            // 4. Add the response to the response context

            // 5. Check to see if this response should be forwarded
            // immediately
            if (newResponse.getStatusCode() == Response.TRYING) {
               return;
            }

            // 6. When necessary, choose the best final response from the

            // 7. Aggregate authorization header fields if necessary

            // 8. Optionally rewrite Record-Route header field values

            // 9. Forward the response
            forwardResponse(stx, newResponse);

            // 10. Generate any necessary CANCEL requests

         } catch (Exception e) {
            e.printStackTrace();
         }
```

4.1.4. Register handler

`REGISTER` request are handled differently. Depending on target domain incoming message is:

- forwarded if target domain does not match configured local domains.
- directed to `ProxySbb` child for processing.

```
public void onRegisterEvent(RequestEvent event, ActivityContextInterface ac) {

    if (logger.isDebugEnabled())
        logger
                .debug("Received REGISTER request, class="
                        + event.getClass());

    // see http://tools.ietf.org/html/rfc3261#section-10.2
    SipURI uri = (SipURI) event.getRequest().getRequestURI();
    if (isRegisterLocal(uri, getProxyConfigurator().getLocalDomainNames())) {
        try {
            ac.attach(getRegistrarSbbChildRelation().create());
        } catch (Exception e) {
            // failed to attach the register, send error back
            logger.error(e);
        }
        // detach myself
        ac.detach(sbbContext.getSbbLocalObject());
    } else {
        processRequest(event.getServerTransaction(), event.getRequest(), ac);
    }

}
```

4.1.5. Child relation

`ProxySbb` relies on its children to perform task on `AOR`, that is store, manage and retrieve bindings. Child relation is declared in SBB descriptor as follows:

```
    <sbb-ref>
        <sbb-name>SipRegistrarSbb</sbb-name>
        <sbb-vendor>org.mobicents</sbb-vendor>
        <sbb-version>1.2</sbb-version>
        <sbb-alias>RegistrarSbb</sbb-alias>
    </sbb-ref>

    ...

    <sbb-classes>
        <sbb-abstract-class>
            <sbb-abstract-class-name>
```

```
            org.mobicents.slee.services.sip.proxy.ProxySbb
        </sbb-abstract-class-name>

        ....

        <get-child-relation-method>
            <sbb-alias-ref>RegistrarSbb</sbb-alias-ref>
            <get-child-relation-method-name>
                getRegistrarSbbChildRelation
            </get-child-relation-method-name>
            <default-priority>0</default-priority>
        </get-child-relation-method>


    </sbb-abstract-class>
</sbb-classes>
```

Parent declaration in descriptor defines method name, in case of example its `getRegistrarSbbChildRelation`, implented by JSLEE. This method allows parent to access `javax.slee.ChildRelation` object representing link to defined child. `ChildRelation` object gives access to `SbbLocalObject` interface. This object allows parent to:

- attach child to `ActivityContextInterface` to make it eligible to receive events.
- invoke synchronously methods defined in child `SbbLocalObject`.

`ProxySbb` invokes its children with both. `RegistrarSbb` is beeing attached to `ActivityContextInterface`. `LocationSbb` is invoked in synchronous way. It exposes `SbbLocalObject` extending `org.mobicents.slee.services.sip.location.LocationService` interface. It is defined and used as follows:

```
public interface LocationSbbLocalObject extends SbbLocalObject,LocationService {

}

public abstract class ProxySbb implements Sbb {

    public abstract ChildRelation getLocationSbbChildRelation();

   public LocationSbbLocalObject getLocationSbb() throws TransactionRequiredLocalException
```

```
, SLEEException, CreateException {
    final ChildRelation childRelation = getLocationSbbChildRelation();
    if (childRelation.isEmpty())
    {
        return (LocationSbbLocalObject) childRelation.create();
    }
    else {
        return (LocationSbbLocalObject) childRelation.iterator().next();
    }

}
...

private void processRequest(ServerTransaction serverTransaction,
        Request request, ActivityContextInterface ac) {

...

    new ProxyMachine(getProxyConfigurator(), getLocationSbb(),
    this.addressFactory, this.headerFactory,
    this.messageFactory, this.provider)
    .processRequest(serverTransaction, request);

...

}

class ProxyMachine extends MessageUtils implements MessageHandlerInterface {
    ...

    protected LocationService reg = null;

    ...

    public ProxyMachine(ProxyConfiguration config,
            LocationService registrarAccess, AddressFactory af,
            HeaderFactory hf, MessageFactory mf, SipProvider prov)
    throws ParseException {
        ...
        reg = registrarAccess;
        ...

    }
```

```
/**
 * Attempts to find a locally registered contact address for the given
 * URI, using the location service interface.
 */
public LinkedList<URI> findLocalTarget(URI uri)
throws SipSendErrorResponseException {
   String addressOfRecord = uri.toString();

   Map<String, RegistrationBinding> bindings = null;
   LinkedList<URI> listOfTargets = new LinkedList<URI>();
   try {
      bindings = reg.getBindings(addressOfRecord);
   } catch (LocationServiceException e) {

      e.printStackTrace();
      return listOfTargets;
   }

   if (bindings == null) {
      throw new SipSendErrorResponseException("User not found",
            Response.NOT_FOUND);
   }
   if (bindings.isEmpty()) {
      throw new SipSendErrorResponseException(
            "User temporarily unavailable",
            Response.TEMPORARILY_UNAVAILABLE);
   }

   Iterator it = bindings.values().iterator();
   URI target = null;
   while (it.hasNext()) {
      String contactAddress = ((RegistrationBinding)it.next()).getContactAddress();
      try {
         listOfTargets.add(af.createURI(contactAddress));
      } catch (ParseException e) {
         log.warn("Ignoring contact address "+contactAddress+" due to parse error",e);
      }
   }
   if (listOfTargets.size() == 0) {

      throw new SipSendErrorResponseException(
            "User temporarily unavailable",
            Response.TEMPORARILY_UNAVAILABLE);
   }
```

```
            return listOfTargets;
        }
    }
}
```

4.2. RegistrarSbb

`RegistrarSbb` is responsible for handling `REGISTER` requests and sending proper response. `RegistrarSbb` receives requests on `ActivityContextInterface` to which it is attached by its parent, please see *Section 4.1.4, "Register handler"* for explanation and code example.

Class `org.mobicents.slee.services.sip.registrar.RegistrarSbb` includes all the service logic required to perform registry tasks.

4.2.1. Register handler

`RegistrarSbb` operate on `RegistrationBindings` managed by `LocationService`.

`RegistrarSbb` performs following operations in `REGISTER` event handler:

- check if request is a query, if so send response with list of contacts

```
// see if child sbb local object is already in CMP
LocationSbbLocalObject locationService = getLocationSbb();

// get configuration from MBean
final long maxExpires=config.getSipRegistrationMaxExpires();
final long minExpires=config.getSipRegistrationMinExpires();

// Process require header

// Authenticate
// Authorize
// OK we're authorized now ;-)

// extract address-of-record
String sipAddressOfRecord = getCanonicalAddress(
(HeaderAddress) event.getRequest().getHeader(ToHeader.NAME));

if (logger.isDebugEnabled()) {
logger.debug("onRegisterEvent: address-of-record from request= " + sipAddressOfRecord);
```

```
    }

    // map will be empty if user not in LS...
    // Note we don't care if the user has a valid account in the LS, we
    // just add them anyway.
    String sipAddress = getCanonicalAddress((HeaderAddress) event.getRequest()
                    .getHeader(ToHeader.NAME));
    Map<String, RegistrationBinding> bindings = locationService
        .getBindings(sipAddress);

    // Do we have any contact header(s)?
    if (event.getRequest().getHeader(ContactHeader.NAME) == null) {
    // Just send OK with current bindings - this request was a
    // query.
    logger.info("query for bindings: sipAddress="+sipAddress);
   sendRegistrationOKResponse(event.getServerTransaction(), event.getRequest(), bindings);
    return;
}
```

- check if request is remove action, if so remove bindings from data base

```
    // Check contact, callid, cseq

    ArrayList newContacts = getContactHeaderList(event.getRequest()
                    .getHeaders(ContactHeader.NAME));
    final String callId = ((CallIdHeader) event.getRequest()
                .getHeader(CallIdHeader.NAME)).getCallId();
    final long seq = ((CSeqHeader) event.getRequest()
                    .getHeader(CSeqHeader.NAME)).getSeqNumber();
    ExpiresHeader expiresHeader = event.getRequest().getExpires();

    if (hasWildCard(newContacts)) { // This is a "Contact: *" "remove
        // all bindings" request
        if ((expiresHeader == null)
                || (expiresHeader.getExpires() != 0)
                || (newContacts.size() > 1)) {
                // malformed request in RFC3261 ch10.3 step 6
            sendErrorResponse(Response.BAD_REQUEST,
                event.getServerTransaction(),event.getRequest());
            return;
        }
```

```
        if (logger.isDebugEnabled()) {
            logger.debug("Removing bindings");
        }
        // Go through list of current bindings
        // if callid doesn't match - remove binding
        // if callid matches and seq greater, remove binding.
        Iterator<RegistrationBinding> it = bindings.values().iterator();

        try {
            while (it.hasNext()) {
                RegistrationBinding binding = (RegistrationBinding) it
                                .next();
                if (callId.equals(binding.getCallId())) {
                    if (seq > binding.getCSeq()) {
                        it.remove();
                        locationService.removeBinding(sipAddressOfRecord,
                                        binding.getContactAddress());
                    } else {
                        sendErrorResponse(Response.BAD_REQUEST,
                        event.getServerTransaction(),event.getRequest());
                        return;
                    }
                } else {
                    it.remove();
                    locationService.removeBinding(sipAddressOfRecord, binding
                            .getContactAddress());
                    }
                }

            } catch (LocationServiceException lse) {
                logger.error(lse);
                sendErrorResponse(Response.SERVER_INTERNAL_ERROR,
                    event.getServerTransaction(),event.getRequest());
                return;
            }

        sendRegistrationOKResponse(event.getServerTransaction(),
            event.getRequest(), bindings);
    }else {
```

- check condition for udpate, update bindings, add,remove and send response

```
}else {
    // Update bindings
    if (logger.isDebugEnabled()) {
        logger.debug("Updating bindings");
    }
    ListIterator li = newContacts.listIterator();

    while (li.hasNext()) {
        ContactHeader contact = (ContactHeader) li.next();

        // get expires value, either in header or default
        // do min-expires etc
        long requestedExpires = 0;

        if (contact.getExpires() >= 0) {
            requestedExpires = contact.getExpires();
        } else if ((expiresHeader != null)
            && (expiresHeader.getExpires() >= 0)) {
            requestedExpires = expiresHeader.getExpires();
        } else {
            requestedExpires = 3600; // default
        }

        // If expires too large, reset to our local max
        if (requestedExpires > maxExpires) {
            requestedExpires = maxExpires;
        } else if ((requestedExpires > 0)
            && (requestedExpires < minExpires)) {
            // requested expiry too short, send response with
            // min-expires
            //
            sendIntervalTooBriefResponse(event.getServerTransaction(),
                    event.getRequest(), minExpires);
            return;
        }

        // Get the q-value (preference) for this binding - default
        // to 0.0 (min)
        float q = 0;
        if (contact.getQValue() != -1)
            q = contact.getQValue();
            if ((q > 1) || (q < 0)) {
```

```
                sendErrorResponse(Response.BAD_REQUEST,
                        event.getServerTransaction(),event.getRequest());
                return;
            }

        // Find existing binding
        String contactAddress = contact.getAddress().getURI().toString();

        RegistrationBinding binding = (RegistrationBinding) bindings
                        .get(contactAddress);

        if (binding != null) { // Update this binding

            if (callId.equals(binding.getCallId())) {
                if (seq <= binding.getCSeq()) {
                    sendErrorResponse(Response.BAD_REQUEST,
                        event.getServerTransaction(),event.getRequest());
                    return;
                }
            }

            if (requestedExpires == 0) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Removing binding: "
                        + sipAddressOfRecord + " -> "
                            + contactAddress);
                }
                bindings.remove(contactAddress);
                locationService.removeBinding(sipAddressOfRecord,
                        binding.getContactAddress());
            } else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Updating binding: "
                        + sipAddressOfRecord + " -> "
                        + contactAddress);
                    logger.debug("contact: " + contact.toString());
                }
                // Lets push it into location service, this will
                // update version of binding
                binding.setCallId(callId);
                binding.setExpires(requestedExpires);
                binding.setRegistrationDate(System.currentTimeMillis());
                binding.setCSeq(seq);
                binding.setQValue(q);
```

```
                locationService.updateBinding(binding);
            }

        } else {
            // Create new binding
            if (requestedExpires != 0) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Adding new binding: "
                    + sipAddressOfRecord + " -> "
                    + contactAddress);
                    logger.debug(contact.toString());
                 }

                    // removed comment parameter to registration binding
                    // - Address and Contact headers don't have comments
                    // in 1.1
                    RegistrationBinding registrationBinding = locationService
                    .addBinding(sipAddress,
                        contactAddress, "",
                        requestedExpires, System.currentTimeMillis(), q, callId,
                        seq);
                   bindings.put(registrationBinding.getContactAddress(),
                            registrationBinding);

                }
            }
                }
            // Update bindings, return 200 if successful, 500 on error
            sendRegistrationOKResponse(event.getServerTransaction(),
                event.getRequest(), bindings);
    }
```

4.2.2. Child relation

`RegistrarSbb` defines `LocationSbb` as child. `LocationSbb` child is used as manager for `RegistrationBindings`. Child relation is declared in sbb-jar.xml descriptor:

```
<sbb-jar>
  <sbb id="sip-registrar-sbb">
        <description>JAIN SIP Registrar SBB</description>
        <sbb-name>SipRegistrarSbb</sbb-name>
```

```
        <sbb-vendor>org.mobicents</sbb-vendor>
        <sbb-version>1.2</sbb-version>

        <sbb-ref>
          <sbb-name>LocationSbb</sbb-name>
          <sbb-vendor>org.mobicents</sbb-vendor>
          <sbb-version>1.2</sbb-version>
          <sbb-alias>LocationSbb</sbb-alias>
        </sbb-ref>


        <sbb-classes>
          <sbb-abstract-class>
            <sbb-abstract-class-name>
              org.mobicents.slee.services.sip.registrar.RegistrarSbb
            </sbb-abstract-class-name>
            <get-child-relation-method>
              <sbb-alias-ref>LocationSbb</sbb-alias-ref>
              <get-child-relation-method-name>
                getLocationSbbChildRelation
              </get-child-relation-method-name>
              <default-priority>0</default-priority>
            </get-child-relation-method>
          </sbb-abstract-class>
        </sbb-classes>

        ...
    </sbb>
</sbb-jar>
```

Parent declaration in descriptor defines method name, in case of example its `getLocationSbbChildRelation`, implented by JSLEE. This method allows parent to access `javax.slee.ChildRelation` object representing link to defined child. `ChildRelation` object gives access to `SbbLocalObject` interface. This object allows parent to:

- attach child to `ActivityContextInterface` to make it eligible to receive events.
- invoke synchronously methods defined in child `SbbLocalObject`.

Its defined in source as follows:

```
public abstract class RegistrarSbb implements Sbb {

    ...

    // location service child relation
    public abstract ChildRelation getLocationSbbChildRelation();

   public LocationSbbLocalObject getLocationSbb() throws TransactionRequiredLocalException
        , SLEEException, CreateException {
        return (LocationSbbLocalObject) getLocationSbbChildRelation().create();
    }
    ...
}
```

4.3. LocationSbb

`LocationSbb` is responsible for storing and managing expiration of address bindings.

Class `org.mobicents.slee.services.sip.location.LocationSbb` includes all the service logic required to perform management of contact addresses.

4.3.1. SBB Interface

`LocationSbb` declares custom `SbbLocalObject` interface. Declared methods are used by `ProxySbb` and `RegistrarSbb` to access registration data. Declaration in sbb-jar.xml look as follows:

```
 <sbb-jar>
   <sbb id="sip-registrar-location-sbb">

      <description>Location Service - JPA based</description>

      <sbb-name>LocationSbb</sbb-name>
      <sbb-vendor>org.mobicents</sbb-vendor>
      <sbb-version>1.2</sbb-version>

      <library-ref>
         <library-name>sip-services-library</library-name>
         <library-vendor>org.mobicents</library-vendor>
         <library-version>1.2</library-version>
      </library-ref>
```

```
      <sbb-classes>
        <sbb-abstract-class>
          <sbb-abstract-class-name>
            org.mobicents.slee.services.sip.location.LocationSbb
          </sbb-abstract-class-name>

        </sbb-abstract-class>
        <sbb-local-interface>
          <sbb-local-interface-name>
            org.mobicents.slee.services.sip.location.LocationSbbLocalObject
          </sbb-local-interface-name>
        </sbb-local-interface>
        ...
      </sbb-classes>

        ...
    </sbb>
</sbb-jar>
```

`org.mobicents.slee.services.sip.location.LocationSbbLocalObject` is defined as follows:

```
package org.mobicents.slee.services.sip.location;

import javax.slee.SbbLocalObject;

public interface LocationSbbLocalObject extends SbbLocalObject,LocationService {

}
```

```
package org.mobicents.slee.services.sip.location;

public interface LocationService extends ... {

    /**
     * Adds new contact binding for particular user..
     *
     * @param sipAddress -
```

```
     *          sip address of record sip:ala@ma.kota.w.domu.com
     * @param contactAddress -
     *          contact address - tel:+381243256
     * @param comment -
     *          possible comment note
     * @param expires -
     *          long - seconds for which this contact is to remain valid
     * @param registrationDate -
     *          long - date when the registration was created/updated
     * @param qValue -
     *          q parameter
     * @param callId -
     *          call id
     * @param cSeq -
     *          seq numbers
     * @return - binding created in this operation
     * @throws LocationServiceException
     */
    public RegistrationBinding addBinding(String sipAddress,
            String contactAddress, String comment, long expires,
            long registrationDate, float qValue, String callId, long cSeq)
            throws LocationServiceException;

    /**
     * Returns set of user that have registered - set contains adress of record
     * for each user, something like sip:ala@kocia.domena.com
     *
     * @return
     * @throws LocationServiceException
     */
    public Set<String> getRegisteredUsers() throws LocationServiceException;

    /**
     * Returns map which contains mapping contactAddress->registrationBinding
     * for particular user - address of record sip:nie@ma.mnie.tu
     *
     * @param sipAddress
     * @return
     * @throws LocationServiceException
     */
    public Map<String, RegistrationBinding> getBindings(String sipAddress)
            throws LocationServiceException;
```

```
        /**
         * Updates the specified registration binding.
         *
         * @param registrationBinding
         * @throws LocationServiceException
         */
        public void updateBinding(RegistrationBinding registrationBinding)
                throws LocationServiceException;

        /**
         * Removes contact address from user bindings.
         *
         * @param sip address of record -
         *          sip:ala@kocia.domena.au
         * @param contactAddress -
         *          tel:+481234567890
         * @throws LocationServiceException
         */
        public void removeBinding(String sipAddress, String contactAddress)
                throws LocationServiceException;


        ...

    }
```

Methods defined by `LocationSbbLocalObject` are implemented by `LocationSbb` class. Those methods are invoked in synchronous manner in order to perform binding operations:

- add binding and create expiration timer

```
public RegistrationBinding addBinding(String sipAddress,
    String contactAddress, String comment, long expires, long registrationDate,
    float qValue, String callId, long cSeq)
    throws LocationServiceException {

                // add binding
    RegistrationBinding registrationBinding = locationService.addBinding(
        sipAddress, contactAddress, comment, expires, registrationDate
        , qValue, callId, cSeq);
    if (logger.isDebugEnabled()) {
```

```
        logger.debug("addBinding: "+registrationBinding);
    }
    // create null aci
    NullActivity nullActivity = nullActivityFactory.createNullActivity();
    ActivityContextInterface aci = null;
    try {
        aci = nullACIFactory.getActivityContextInterface(nullActivity);
        // set name
        activityContextNamingFacility.bind(aci, getACIName(contactAddress, sipAddress));
    } catch (Exception e) {
        throw new LocationServiceException(e.getLocalizedMessage());
    }
    // atach to this activity
    aci.attach(sbbContext.getSbbLocalObject());
    // set timer
    TimerID timerID = timerFacility.setTimer(aci, null,registrationDate +
        ((expires+1)*1000), defaultTimerOptions);
    // save data in aci
    RegistrationBindingActivityContextInterface rgAci = asSbbActivityContextInterface(aci);
    rgAci.setTimerID(timerID);
    rgAci.setContactAddress(contactAddress);
    rgAci.setSipAddress(sipAddress);

    if(logger.isInfoEnabled()) {
        logger.info("added binding: sipAddress="+sipAddress+"
            ,contactAddress="+contactAddress);
    }

    return registrationBinding;
}
```

- update binding and restart expiration timer

```
public void updateBinding(RegistrationBinding registrationBinding)
    throws LocationServiceException {

    if (logger.isDebugEnabled()) {
        logger.debug("updateBinding: registrationBinding="+registrationBinding);
    }

    // get named aci
```

```
    ActivityContextInterface aci = activityContextNamingFacility.lookup(
        getACIName(registrationBinding.getContactAddress()
        , registrationBinding.getSipAddress()));
    // get the timer id from the aci and reset the timer
    RegistrationBindingActivityContextInterface rgAci = asSbbActivityContextInterface(aci);
    timerFacility.cancelTimer(rgAci.getTimerID());
    rgAci.setTimerID(timerFacility.setTimer(aci, null
        , registrationBinding.getRegistrationDate()
        + ((registrationBinding.getExpires()+1) * 1000), defaultTimerOptions));
    // update in location service
    locationService.updateBinding(registrationBinding);

    if(logger.isInfoEnabled()) {
        logger.info("binding updated: sipAddress="+registrationBinding.getSipAddress()
        +",contactAddress="+registrationBinding.getContactAddress());
    }
}
```

- remove binding and cancel expiration timer

```
public void removeBinding(String sipAddress, String contactAddress)
    throws LocationServiceException {

    if (logger.isDebugEnabled()) {
        logger.debug("removeBinding: sipAddress="+sipAddress+"
            ,contactAddress="+contactAddress);
    }

    try {
        // lookup null aci from aci naming facility, get timerid and cancel
        // timer (when present).
        ActivityContextInterface aci = activityContextNamingFacility
            .lookup(getACIName(contactAddress, sipAddress));
        if (aci != null) {
            timerFacility.cancelTimer(asSbbActivityContextInterface(aci)
                        .getTimerID());
            activityContextNamingFacility.unbind(getACIName(contactAddress, sipAddress));
            // end null activity, detach is no good because this is a different
            // sbb entity then the one that create the binding
            ((NullActivity)aci.getActivity()).endActivity();
        }
```

```
    } catch (Exception e) {
        throw new LocationServiceException(e.getLocalizedMessage());
    }
    // remove from location service
    locationService.removeBinding(sipAddress, contactAddress);

    if(logger.isInfoEnabled()) {
        logger.info("removed binding: sipAddress="+sipAddress+"
            ,contactAddress="+contactAddress);
    }
}
```

- get registered users AOR

```
    public Set<String> getRegisteredUsers() throws LocationServiceException {

        if (logger.isDebugEnabled()) {
            logger.debug("getRegisteredUsers");
        }

        return locationService.getRegisteredUsers();
    }
```

- retrieve all bindings for AOR

```
    public Map<String, RegistrationBinding> getBindings(String sipAddress)
            throws LocationServiceException {

        if (logger.isDebugEnabled()) {
            logger.debug("getBindings: sipAddress="+sipAddress);
        }

        return locationService.getBindings(sipAddress);
    }
```

4.3.2. SBB Activity Context Interface

`LocationSbb` defines custom `Activity Context Interface` with variables. Variables are used to store data required to manage expiration timer and access binding in storage, that is:

- JSLEE timer id of timer running for contact address
- contact address for which expiration timer runs
- sip address (AOR) of binding

Custom `Activity Context Interface` is defined in sbb-jar.xml descriptor as follows:

```
<sbb id="sip-registrar-location-sbb">

      <description>Location Service - JPA based</description>

      <sbb-name>LocationSbb</sbb-name>
      <sbb-vendor>org.mobicents</sbb-vendor>
      <sbb-version>1.2</sbb-version>

      <library-ref>
         <library-name>sip-services-library</library-name>
         <library-vendor>org.mobicents</library-vendor>
         <library-version>1.2</library-version>
      </library-ref>

      <sbb-classes>
         <sbb-abstract-class>
            <sbb-abstract-class-name>
               org.mobicents.slee.services.sip.location.LocationSbb
            </sbb-abstract-class-name>

         </sbb-abstract-class>
         ...
         <sbb-activity-context-interface>
            <sbb-activity-context-interface-name>
               org.mobicents.slee.services.sip.location.RegistrationBindingActivityContextInterface
            </sbb-activity-context-interface-name>
         </sbb-activity-context-interface>
      </sbb-classes>
      ...
   </sbb>
</sbb-jar>
```

Custom `Activity Context Interface` accessor is defined as follows:

```
public abstract class LocationSbb implements ... {
    public abstract RegistrationBindingActivityContextInterface asSbbActivityContextInterface(
            ActivityContextInterface aci);
}
```

Variables are defined in `Java Bean` convetion, by declaration of setter and getter pair in custom `Activity Context Interface`:

```
public interface RegistrationBindingActivityContextInterface extends ActivityContextInterface {

    public abstract TimerID getTimerID();
    public abstract void setTimerID(TimerID timerID);

    public abstract String getContactAddress();
    public abstract void setContactAddress(String contactAddress);

    public abstract String getSipAddress();
    public abstract void setSipAddress(String sipAddress);

}
```

4.3.3. Expiration timer

Timer supervising expiration is created with binding. Its attached to named ACI. Timeout value is based on data passed from `RegistrarSbb`.

Timer event handler is invoked in case of contact address expiration, its purpose is to:

- retrieve data stored in custom `Activity Context Interface`
- clean ACI name binding
- remove AOR binding from `LocationService`

Timer event handler is defined as follows:

```
public void onTimerEvent(TimerEvent timer, ActivityContextInterface aci) {

    if (logger.isFineEnabled()) {
        logger.fine("onTimerEvent()");
    }

    aci.detach(sbbContext.getSbbLocalObject());

    // cast to rg aci
    RegistrationBindingActivityContextInterface rgAci = asSbbActivityContextInterface(aci);
    // get data from aci
    String contactAddress = rgAci.getContactAddress();
    String sipAddress = rgAci.getSipAddress();

    // unbind from aci so it ends
    try {
        activityContextNamingFacility.unbind(getACIName(contactAddress, sipAddress));
    } catch (Exception e) {
        logger.severe("",e);
    }
    // remove rg from location service
    try {
        locationService.removeBinding(sipAddress, contactAddress);
    } catch (Exception e) {
        logger.severe("",e);
    }

    if(logger.isInfoEnabled()) {
        logger.info("binding expired: sipAddress="+sipAddress+"
            ,contactAddress="+contactAddress);
    }

}
```

4.3.4. SBB Activity Context Interface naming

`LocationSbb` associates name to ACI on which expiration timer runs. This makes ACI accessible with given name. Name and ACI association is created with JSLEE `Activity Context Naming Facility`. Name uniquely identifies `Timer` ACI for given AOR and contact address.

4.3.5. SBB environment entries

`LocationSbb` environment entry controls `LocationService` type used(JPA and local). XML descriptor defines entry as follows:

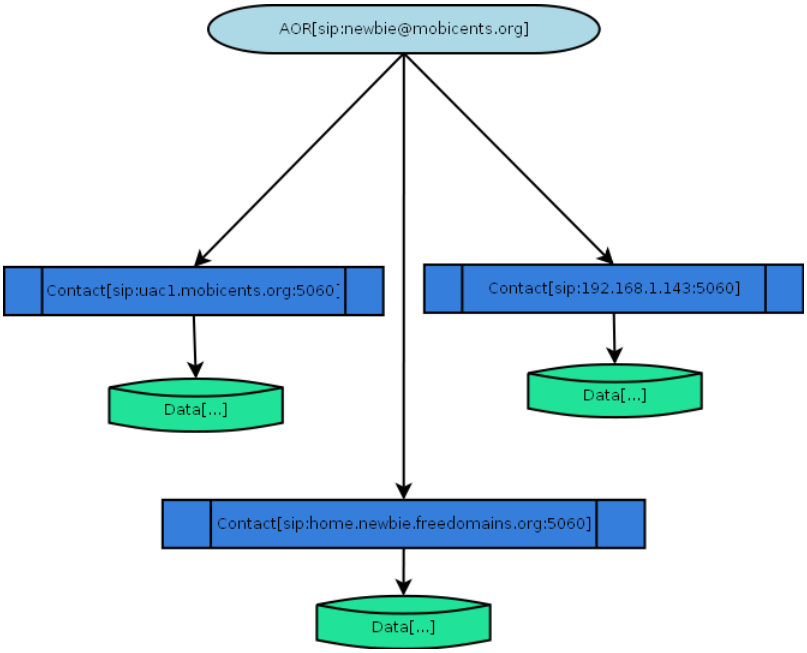
```
<sbb id="sip-registrar-location-sbb">

      <description>Location Service - JPA based</description>

      <sbb-name>LocationSbb</sbb-name>
      <sbb-vendor>org.mobicents</sbb-vendor>
      <sbb-version>1.2</sbb-version>

      <library-ref>
         <library-name>sip-services-library</library-name>
         <library-vendor>org.mobicents</library-vendor>
         <library-version>1.2</library-version>
      </library-ref>

      ...
      <env-entry>
         <env-entry-name>LOCATION_SERVICE_CLASS_NAME</env-entry-name>
         <env-entry-type>java.lang.String</env-entry-type>
         <!-- choose your location service, filtered on compliation -->
         <env-entry-value>
            org.mobicents.slee.services.sip.location.nonha.NonHALocationService
         </env-entry-value>
      </env-entry>
   </sbb>
</sbb-jar>
```

Environment entries are accessible with JNDI lookups:

```
      Context myEnv = (Context) new InitialContext().lookup("java:comp/env");
      String value  = (String) myEnv.lookup("LOCATION_SERVICE_CLASS_NAME");
```

4.4. Location Service

Location service is simple POJO based service. Its only purpose is to persist bindings. It stores data for each AOR in following form:



Sip Service Example location information structure

Running the Example

The easiest way to test example is to deploy example and start container, register two UACs and make call from one to another. UACs must be configured to use container as proxy, by default each UAC should point to proxy at `127.0.0.1:5060` and have domain set to either `nist.gov` or `mobicents.org`

5.1. Configuration

Two components can be configured in example:

LocationSbb
: location components supports two different storage providers: HA and local. Default storage is local. It can be changed with sbb-env property: `LOCATION_SERVICE_CLASS_NAME` . It can have one of two values:

  org.mobicents.slee.services.sip.location.nonha.NonHALocationService
  : this class implements local service, not replicated, based on java `java.util.Map`.

  org.mobicents.slee.services.sip.location.jpa.JPALocationService
  : this class implements HA service, replicated storage, based on JPA.

  `LocationSbb` exposes JMX properties which can be accessed under `ObjectName: slee:sipservice=Location` . Supported properties:

Table 5.1. LocationSbb JMX properties

Name	Type	Description
registeredUsers	java.util.Set<java.lang.String>	Read only property. It return value is set with AORs of users.
registeredUserCount	java.lang.Integer	Read only property. Indicates how many users are registered.

ProxySbb
: Configuration can be changed in `configuration.properties` file or during runtime with JMX Bean. Bean is accessible under `ObjectName: slee:sipproxyconfigurator=only_human` , where ***`sipproxyconfigurator`*** value is equal to configuration name. Supported properties:

Table 5.2. ProxySbb JMX properties

Name	Type	Description
supportedURIScheme	java.lang.String	URI scheme supported by proxy. It coma separated list.

Name	Type	Description
		By default it has value of {sip,sips}
localDomain	java.lang.String	domain for which this proxy should provide services. Its comma separated list of values. By default it has value of {127.0.0.1, nist.gov ,mobicents.org}
transports	java.lang.String	list of transports which can be used by proxy. Its comma separated list of values. By default it has value of {udp}, by default proxy uses first one available.
sipPort	java.lang.Integer	Port which is used to send and receive messages. By defautl it has value of {5060}

RegistrarSbb

Configuration can be changed during runtime with JMX Bean. Bean is accessible under `ObjectName: slee:sipregistrarconfigurator=v1RegistrarConf` . Supported properties:

Table 5.3. RegistrarSbb JMX properties

Name	Type	Description
sipRegistrationMinExpires	java.lang.Integer	Minimal acceptable value of `Expires` Header
sipRegistrationMaxExpires	java.lang.Integer	Maximal acceptable value of `Expires` Header

Traces and Alarms

6.1. Tracers

The example Application uses multiple JAIN SLEE 1.1 Tracer facility instances. Below is full list:

Table 6.1. Sip Service Tracer and Log Categories

Sbb	Tracer name	LOG4J category
ProxySbb	ProxySbb	javax.slee.SbbNotification[service= ServiceID[name=JAIN SIP Proxy Service,vendor=mobicents, version=1.1],sbb=SbbID[name=ProxySbb, vendor=mobicents,version=1.1]] .ProxySbb
RegistrarSbb	RegistrarSbb	javax.slee.SbbNotification[service= ServiceID[name=SIP Registrar Service,vendor=org.mobicents, version=1.2],sbb=SbbID[name=SipRegistrarSbb, vendor=org.mobicents,version=1.2]] .RegisterSbb
LocationSbb	LocationSbb	javax.slee.SbbNotification[service= ServiceID[name=SIP Registrar Service,vendor=org.mobicents, version=1.2],sbb=SbbID[name=LocationSbb, vendor=org.mobicents,version=1.2]] .LocationSbb

Important

Spaces where introduced in `LOG4J category` column values, to correctly render the table. Please remove them when using copy/paste.

6.2. Alarms

The example Application does not sets JAIN SLEE Alarms.

Appendix A. Revision History

Revision History
Revision 1.0 Mon Feb 8 2010 BartoszBaranowski
Creation of the JBoss Communications JAIN SLEE Sip Service Example User Guide.

Index

F