Narval - Jabber how-to

Edition: Version 2 - 2004-12-15

Sylvain Thénault

Narval - Jabber how-to

Sylvain Thénault

Edition: Version 2 - 2004-12-15

Table of Content

Chapter I

The Jabber protocol handler

The Jabber protocol is handled in Narval by a component called a *protocol handler*. A protocol handler is an object with a particular interface providing a way to dynamically plug into the Narval interpreter some support for different (usually network) communication protocols. When it receives a message, a protocol handler will format that message into a narval memory element and start a recipe using this element as its initial context.

A protocol handler is activated by a particular element, its *activator*. To activate the Jabber protocol handler, you'll have to put in your memory file (usually **$NARVAL_HOME/data/memory.xml**, where NARVAL_HOME is by default the .narval directory in your home directory) an element like the following:

```
<jabber-activator host="jabber.logilab.org" port="5222"
                  user="narval" password="narval" resource="narvabber"
                  register="yes" verbose="yes"
```

Notice that by default on standard distribution, the memory file include the **custom.xml** file with a jabber activator element inside it that you only have to adapt to your settings.

Of course, you'll have to change some attributes to suit your needs. Here is the description of the different possible attribute on the Jabber's activator:

user
: the Jabber user's name

password
: the Jabber user's password

resource
: the Jabber resource (default to narvabber)

host
: the Jabber server host name (default to localhost)

port
: the Jabber server port (default to 5222)

register
: indicates whether the agent should try to auto-register to the server if necessary, i.e. if the narval's account isn't existing and should be created ('yes' or 'no', default to 'no')

verbose
: indicates whether the handler should log some additional information about information it receives and things it is doing ('yes' or 'no', default to 'no')

recipe
: the full name of the recipe (<cookbook>.<recipe>) used to handle incoming queries

So you should at least change the *user* and *password* attributes, and also probably the *host* attribute, unless you're usually using Logilab's Jabber server (which is for private use). Only the user, password and recipe attributes are required, others will take a default value if not specified.

The following sections explains how to get an extensible chat bot using Narval's actions and recipes, and how to control / extend it. Of course, you can also create your own recipe for your specific needs!

Chapter II

The default Jabber processing mechanism in Narval

The chat bot is currently built using the following principle:

- a recipe, 'bots.handle-commands' is handling incoming messages (that is the target of the "recipe" attribute on the Jabber activator), and produces some *command elements* according to an ail [http://www.logilab.org/projects/ail] rules file (by default **$NARVAL_HOME/data/chatbot.ail**, see Modifying the ail rules file section for description of the syntax used there) and to the bot-configuration element which is handling right access to the existing commands (see Configuring commands access right using the <bot-configuration> element section)
- a set of *plans* are activated on Narval's startup by adding several <al:start-plan> elements to the memory file. A plan is an active execution of a recipe. These plans are particular for they are written to indefinitely wait for a specific type of element to be produced. When such an element is produced, the plan is resumed to process it. As it holds a restart tag, another instance of the same recipe is created when the plan reaches its end. In the case of the chat bot, we usually have one active plan per possible command element. Currently most of the chat bot command handling recipes are in the *active-commands* cookbook.
- the Jabber protocol handler is waiting for the production of message or presence element with a type == 'outgoing', and automatically send them to the Jabber server when they are produced.

The main advantages of this architecture are the following :

- you have a set of simple recipes instead of a giant one
- you don't have to edit the handler's recipe to add new functionnalities
- you can have multiple recipes handling a single command
- if you do not want to use a command, you just have to remove the corresponding start-plan element

The main drawback is that to handle a new command, you usually have to change / add stuff in multiple places, as explained in the section below.

Chapter III

How to handle a new command

Let us say we want to add a new command to the chat bot, "beep". When it receives this in a Jabber message, your assistant should run the "beep" command on the host where it runs ("beep" is a Unix command ringing your computer's internal bell).

1. Creating the "beep" recipe

It is very simple to create the recipe since we have an action which is able to run arbitrary command on the system, *Basic.command*. We mainly have to wrap it in a recipe and to tell it to run the "beep" command. So let us edit a file with your personnal recipes, say $NARVAL_HOME/recipes/myrecipes.xml (you can change the name of the cookbook file, but its place should be **$NARVAL_HOME/recipes** so it will be automatically loaded on Narval's startup).

We end with the following content :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<al:cookbook xmlns:al="http://www.logilab.org/namespaces/Narval/1.2">
<al:recipe name="beep">
  <al:step id="1" type="action" target="Basic.command">
    <al:arguments>
      <data>beep</data>
    </al:arguments>
  </al:step>
</al:recipe>
```

We have a recipe named *beep*, made of a single step which target is the *Basic.command* action. If we take a look at this action's prototype in the documentation, we can see that it takes a <data> element containing the command to run as input. In the above case, since the element is a constant, it can be given using the <al:arguments> child element of the step definition.

You can test this recipe by adding this line in your memory file:

```

```

If you already have a narval running with an active narval-start-plan, you can tell it to start the myrecipe.beep plan by sending it a jabber message saying "start-plan myrecipe.beep".

In the first case, your agent should make your system beep at some point of its startup, in the second case, it should make your system beep when you tell him to start your recipe.

2. Making an "active" recipe using it

We now have a recipe doing what we wanted, and we would like to make an active recipe from it, reacting to a "beep" command that should be produced in some conditions defined in the next section. To make a plan wait indefinitly for an element, you have to add a condition on a transition matching the desired element. So we end with the following recipe in our *myrecipe* cookbook:

```
<al:recipe name="active-beep" restart="yes">
```

```
  <al:step id="1" type="action" target="Basic.nop"/>
  <al:step id="2" type="recipe" target="myrecipes.beep"/>
  <al:transition id="t0">
    <al:in idref="s1"/>
    <al:out idref="s2"/>
    <al:condition use="yes">
      <al:match>ICommand(elmt).name == 'beep'</al:match>
    </al:condition>
  </al:transition>
```

We have a first step doing nothing (*nop* stands here for "no operation") and then a transition to a step running the recipe created in the previous section. On this transition, we have a condition telling that we are waiting for an element implementing the *ICommand* interface, and that the name attribute of this command element should have the "beep" value. The *restart* attribute is set to "yes" so that the plan is automatically restarted when it ends.

3. Modifying the ail rules file

The following step is to modify the rules file to produce this command element on some user input. To do so we can add the following lines to it:

```
\s*RING|BOP|SONNE\s* >> BEEP
```

There are two types of rules in the ail syntax, *rewrite* rules and *command* (final) rules. A rule is defined by two parts separated by a *::* for command rules, or a *>>* for rewrite rules. The left-hand side is always a regular expression used to match some input or rewritten text, and the right-hand side is either:

- (for *rewrite* rules) a string used to rewrite the input, which can contain string substitution taken from the input text using the usual regular expression syntax (i.e. "1" will be replaced by the content of the first group of the regular expression in the left part of the rule)
- (for *command* rules) a command definition, made of a function name and some optional arguments, separated by spaces (please note we are not talking about the same commands as the one we have been describing so far). In Narval, you have three functions available:
  - ignore: take no arguments and do nothing
  - random: take arbitrary list of arguments and do create a command element with name == 'response' and as argument one of the input argument choosen randomly
  - command: create a command element whose name is the first argument and following arguments are included as the command element's arguments

  The command definition can also use string substitutions as rewrites do.

It is to be noticed that regular expressions are compiled so that they are case insensitive. When processing some input text, the algorithm is:

1. take rules in order until a left-hand side regular expression is found that matches the input text
2. depending on the type of the matching rule:

   •

```
(rewrite rule) rewrite the text using the right-hand side of the rule
and
```

   go back to 1) with the rewritten text as input

   - (command rule) call the command function using given arguments and stop there
3. if no rule matches, raise an error.

So in our case, we have added:

- a rewrite rule so that "RING", "BOP" and "SONNE" with any space before or after get rewritten to "BEEP"
- a command rule so that "BEEP" as input produce a command element with 'beep' as name

4. Testing the chat command

We are almost done ! Let us suppose that you have your agent already running. The first thing to do to activate our new command is to start the active plan handling it, by sending the following message to the agent:

Now, you should be able to make your agent ring your computer's bell by just typing for instance "beep" or "ring" in its Jabber window.

5. Modifying the memory file to handle this command permanently

Now that everything works fine, you just have to add the following element to your memory file:

By doing so, this recipe will be automatically started on the next startup.

One point we've not seen yet is how to control who has access to which command. This is explained in the next section.

6. Configuring commands access right using the <bot-configuration> element

The problem is that you don't want anyone to be able to give arbitrary command to your assistant. That is not a big deal for our "beep" command, but it may be for other command, such as "shutdown" for example which stops the agent.

An important attribute of the bot configuration element is *myuser*. This attribute should indicates to the agent the Jabber id of its master. The default command policy (i.e. access to commands with no explicit access right defined) depends on this attribute: * if it is defined, only this user has access to the command * if it is not defined, all users have access to the command

Explicit access rights are defined using <access-right> children elements on the bot configuration. For instance:

```
<bot-configuration myuser="syt">
   <!--
     Commands access rights, given to explicit Jabber user id or to a group
     using one of the following special keywords:
        * all (everybody)
        * myuser (user of the agent, defined on the bot-configuration element)
        * agents (narval agents) NOT YET IMPLEMENTED
        * people (non narval agents) NOT YET IMPLEMENTED
```

```
      You can specify multiple ids separarated by commas. Commands with no
      access right defined default to "myuser" if it is defined, else to "all".
      -->
   <access-right command="shutdown">myuser</access-right>
   <access-right command="response">all</access-right>
   <access-right command="kb_add_user_info">all</access-right>
   <access-right command="kb_add_stmt">myuser, gizmo</access-right>
```

With the previous bot configuration element in my memory file, I have:

- restricted access to the shutdown command to the agent's master, whose Jabber identifier is "syt" (i.e. the value of the "myuser" attribute)
- given access to any user for the "response" and "kb_add_user_info" commands
- restricted access to the "kb_add_stmt" command to both the agent's master and the user with the "gizmo" Jabber id.

Chapter IV

Some other configuration possibilities for existing commands

You can use the following to configure some parts of the chat bot or handled commands.

1. Base configuration

The following element allows for arbitrary location for the ail rules file:

```
<url address="file:$NARVAL_HOME/data/chatbot.ail"
```

2. Logging control

Several parameters influence the logging behviour when observing forum:

- the *min_threshold* and *max_threshold* attributes of the <bot-configuration> element are used to control the note-taking categorization : when it tries to categorize a sentence, the bot will get a confidence level between 0 and 1 (0 means "ignore", 1 means "log"). If this level is lesser than the min_treshold, the sentence will be ignored. If it is greater than the max_treshold, the sentence will be logged. In between, it will ask for feedback.
- the following elements:

```
<url address="file:$NARVAL_HOME/data/bot_log"
     type:name="uri:memory:discussion-log-base" encoding="iso-8859-1"/>
<url address="file:$NARVAL_HOME/data/bot_learned_data"
```

  are defining a directory where log files (one for each discussion) and the learned data file will be located

3. Knowledge management

Depending on your knowledge base backend (currently there is one using Pylog [http://christophe.delord.free.fr/en/pylog/] and another one using Redland [http://librdf.org/], you may have to use the following elements to locate either the Pylog file (using a Prolog [http://gprolog.inria.fr/] syntax) or the Redland file (using XML/RDF [http://www.logilab.org/projects/ail] syntax). Note that both could be used in parallel as they have different capabilities.

```
<url address="file:$NARVAL_HOME/data/kb.pl"
     type:name="uri:memory:kb"/>
<url address="file:$NARVAL_HOME/data/rdfstore.rdf"
```

Finally the following is giving the address of your personal FOAF [http://www.foaf-project.org/] file:

```
<url address="file:$NARVAL_HOME/data/foaf.rdf"
```

Chapter V

Going further

To go further with Narval, you should read the narval user [http://www.logilab.org/projects/narval/doc/user_manual] and developper [http://www.logilab.org/projects/narval/doc/developper_manual] manuals. You can also join the ai-projects [http://www.logilab.org/mailinglists/ai_projects] mailing-list in order to discuss with users and developers.