The Ivy java library guide

CENA NT02-819

Yannick Jestin

jestin@cena.fr

This document is a programmer's guide that describes how to use the Ivy Java library to connect applications to an Ivy bus. This guide describes version 1.2.6 of the library. This document itself is part of the java package, available on the Ivy web site (http://www.tls.cena.fr/products/ivy/).

1. Foreword

This document was written in SGML according to the DocBook DtD, so as to be able to generate PDF and html output. However, the authors have not yet mastered the intricacies of SGML, the DocBook DtD, the DocBook Stylesheets and the related tools, which have achieved the glorious feat of being far more complex than LaTeX and Microsoft Word combined together. This explains why this document, in addition to being incomplete, is quite ugly. We'll try and improve it.

2. What is Ivy?

Ivy is a software bus designed at CENA (http://www.cena.fr/). A software bus is a system that allows software applications to exchange information with the illusion of broadcasting that information, selection being performed by the receiving applications. Using a software bus is very similar to dealing with events in a graphical toolkit: on one side, messages are emitted without caring about who will handle them, and on the other side, one decide to handle the messages that have a certain type or follow a certain pattern. Software buses are mainly aimed at facilitating the rapid development of new agents, and at managing a dynamic collection of agents on the bus: agents show up, emit messages and receive some, then leave the bus without blocking the others.

Ivy is implemented as a collection of libaries for several languages and platforms. If you want to read more about the principles Ivy before reading this guide of the java library, please refer to *The Ivy sofware bus: a white paper*. If you want more details about the internals of Ivy, have a look at *The Ivy architecture and protocol*. And finally, if you are more interested in other languages, refer to other guides such as *The Ivy Perl library guide* (not yet written), or *The Ivy C library guide*. All those documents should be available from the Ivy Web site (http://www.tls.cena.fr/products/ivy/).

3. The Ivy java library

3.1. What is it?

The Ivy java library (aka libivy-java or fr.dgac.ivy) is a java package that allows you to connect applications to an Ivy bus. You can use it to write applications in java. You can also use it to integrate any thread-safe java application on an Ivy bus. So far, this library has been tested and used on a variety of java virtual machines (from 1.1.7 to 1.4.2), and on a variety of architectures (GNU/Linux, Solaris, Windows NT,XP,2000, MacOSX).

The Ivy java library was originally developed by François-Régis Colin and is now maintained by Yannick Jestin at CENA within a group at CENA (Toulouse, France).

3.2. Getting and installing the Ivy Java library

You can get the latest versions of the Ivy C library from the Ivy web site (http://www.tls.cena.fr/products/ivy/). It is packaged either as a jar file or as a debian package. We plan to package it according to different distribution formats, such as .msi (Windows) or .rpm (Redhat and Mandrake linux). Contributors are welcome for package management.

The package is mainly distributed as a jar file. In order to use it, either add it in your CLASSPATH environment variable, put the it in your $JAVA_HOME/jre/lib/ext/ directory, or C:\Program Files\JavaSoft\... for Windows. The best way to avoid mistakes is to put it in the command line each time you want to use ivy **$ java -classpath .:/path/to/ivy.jar:/path/to/regexp.jar:/path/to/getopt.jar className**

The package contains the documentation, the sources and the class files for the fr.dgac.ivy package, alongside with examples and a couple of useful tools, IvyDaemon and Probe. You will need the Apache Jakarta project regexp library (http://jakarta.apache.org/regexp/) and the gnu getopt library (http://www.urbanophile.com/arenn/coding/download.html). Those could be included in the jar file, but not in the debian package.

In order to test the presence of Ivy on your system once installed, run the following command:

```
$ java fr.dgac.ivy.Probe
```

If should display a line about broadcasting on a strange address, this is OK and means it is ready and working. If it complains about a missing class (java.lang.NoClassDefFoundError), then you have not pointed your virtual machine to the jar file or your installation is incomplete.

4. Your first Ivy application

We are going to write a "Hello world translater" for an Ivy bus. The application will subscribe to all messages starting with the "Hello" string, and re-emit them on the bus having translated "Hello" into "Bonjour" (Hello in french). In addition, the application will quit as soon as it receives a "Bye" message.

4.1. The code

Here is the code of "ivyTranslater.java":

```
import fr.dgac.ivy.* ;

class ivyTranslater implements IvyMessageListener {

  private Ivy bus;

  ivyTranslater() throws IvyException {
    // initialization, name and ready message
    bus = new Ivy("IvyTranslater","IvyTranslater Ready",null);
    // classical subscription
    bus.bindMsg("^Hello(.*)",this);
    // inner class subscription ( think awt )
    bus.bindMsg("^Bye$",new IvyMessageListener() {
      public void receive(IvyClient client, String[] args) {
// leaves the bus, and as it is the only thread, quits
        bus.stop();
      }
    });
    bus.start(null); // starts the bus on the default domain
  }

  // callback associated to the "Hello" messages"
  public void receive(IvyClient client, String[] args) {
    try {
      bus.sendMsg("Bonjour"+((args.length>0)?args[0]:""));
    } catch (IvyException ie) {
      System.out.println("can't send my message on the bus");
    }
  }

  public static void main(String args[]) throws IvyException {
    new ivyTranslater();
  }
}
```

4.2. Compiling it

You should be able to compile the application with the following command (if the ivy-java jar is in your development classpath):

```
$ javac ivyTranslater.java
$
```

4.3. Testing

We are going to test our application with **fr.dgac.ivy.Probe**. In a shell, launch ivyTranslater:

```
$ java ivyTranslater
```

In another shell, launch **java fr.dgac.ivy.Probe '(.*)'**. You can see that the IvyTranslater has joined the bus, published its subscriptions, and sent the mandatory ready message. As your probe has subscribed to the eager regexp .* and reports the matched string within the brackets (.*), the ready message is printed.

```
$ java fr.dgac.ivy.Probe '(.*)'
you want to subscribe to (.*)
broadcasting on 127.255.255.255:2010
IvyTranslater connected
IvyTranslater subscribes to ^Bye$
IvyTranslater subscribes to ^Hello(.*)
IvyTranslater sent 'IvyTranslater Ready'
```

Probe is an interactive program. Type "Hello Paul", and you should receive "Bonjour Paul". Type "Bye", and the ivyTranslater application should quit to the shell. Just quit Probe, issuing a Control-D (or .quit) on a line, and Probe exists to the shell.

```
Hello Paul
-> Sent to 1 peers
IvyTranslater sent 'Bonjour Paul'
Bye
-> Sent to 1 peers
IvyTranslater disconnected
<Ctrl-D>
$
```

5. Basic functions

The javadoc generated files are available on line on the ivy web site, and should be included in your ivy java package (or in /usr/share/doc/libivy-java, alongside with this very manual). Here are more details on those functions.

5.1. Initialization an Ivy object and joining the bus

Initializing a java Ivy agent is a two step process. First of all, you must create an `fr.dgac.ivy.Ivy` object. It will be the repository of your agent name, network state, subscriptions, etc. Once this object is created, you can subscribe to the various Ivy events: text messages through perl compatible regular expressions, other agents' arrival, departure, subscription or unsubscription to regexps, direct messages or die command issued by other agents. At this point, your ivy application is still not connected. In order to join the bus, call the `start(string domain)` method on your Ivy object. This will spawn two threads that will remain active until you call the `stop()` method on your Ivy object or until some other agent sends you a die message. Once this `start()`

method has been called, the network machinery is set up according to the ivy protocol, and your agent is ready to handle messages on the bus !

```
fr.dgac.ivy.Ivy(String name,String message, IvyApplicationListener appcb)
```

This constructor readies the structures for the software bus connexion. It is possible to have more than one bus at the same time in an application, be it on the same ivy broadcast address or one different ones. The *`name`* is the name of the application on the bus, and will by transmitted to other application, and possibly be used by them (through `String IvyClient.getApplicationName()`). The *`message`* is the first message that will be sent to other applications, with a slightly different broadcasting scheme than the normal one (see *The Ivy architecture and procotol* document for more information. If *`message`* is null, nothing will be sent. Usually, other application subscribe to this ready message to trigger actions depending on the presence of your agent on the bus. The *`appcb`* is an object implementing the IvyApplicationListener interface. Its different methods are called upon arrival or departure of other agents on the bus, or when your application itself leaves the bus, or when a direct message is sent to your application. It is also possible to add or remove other application listeners using the `Ivy.AddApplicationListener()` and `Ivy.RemoveApplicationListener()` functions.

```
public void start(String domainbus) throws IvyException
```

This method connects the Ivy bus to a domain or list of domains. This spawns network managing threads that will be stropped with `Ivy.stop()` or when a die message is received. The rendez-vous point is the String parameter *`domainbus`*, an UDP broadcast address like "10.0.0:1234" (255 are added at the end to become an IPv4 UDP broadcast address). This will determine the meeting point of the different applications. For the gory details, this is done with an UDP broadcast or an IP Multicast, so beware of routing problems ! You can also use a comma separated list of domains, for instance "10.0.0.1234,192.168:3456". If the domain is *`null`*, the API will check for the property *`IVY_BUS`* (set at the invocation of the JVM, e.g $ java -DIVY_BUS=10:4567 myApp, or via an environment variable on older JVMs); if not present, it will use the default bus, which is 127.255.255.255:2010. The default address requires a broadcast enabled loopback interface to be active on your system (CAUTION, on MacOSX and some releases of SunOS, the default bus doesn't work ...). If an IvyException is thrown, your application is not able to send or receive data on the specified domain.

```
public void stop()
```

This methods stops the threads, closes the sockets and performs some clean-up. If there is no other thread running, the program quits. This is the preferred way to quit a program within a callback (please don't use `System.exit()` before having stopped the bus, even if it works ...). Note that it is still possible to reconnect to the bus by calling `start()` once again.

5.2. Emitting messages

Emitting a message is much like echoing a string on a output channel. Portion of the message will be sent to the connected agent if the message matches their subscriptions.

```
public int sendMsg(String message)
```

Will send each remote agent the substrings in case there is a regexp matching. The default behaviour is not to send the message to oneself ! The result is the number of messages actually sent. The main issue here is that the sender ivy agent is the one who takes care of the regexp matching, so that only useful information are conveyed on the network. Be sure that the message sent doesn't contains protocol characters: 0x01 to 0x08 and unfortunately 0x0D, the newline character. If you want to send newlines, see protectNewline, in advanced functions.

5.3. Subscription to messages

Subscribing to messages consists in binding a callback function to a message pattern. Patterns are described by regular expressions with captures. Since ivy-java 1.2.4, Perl Compatible Regular Expressions are used, with the Apache Jakarta Project regexp library (see the jakarta regexp web site (http://jakarta.apache.org/regexp/)). When a message matching the regular expression is detected on the bus (the matching is done at the sender's side), the recipient's callback function is called. The captures (ie the bits of the message that match the parts of regular expression delimited by brackets) are passed to the callback function much like options are passed to main. Use the `bindMsg()` method to bind a callback to a pattern, and the `unbindMsg` method to delete the binding.

```
public int bindMsg(String regexp, IvyMessageListener callback);
public void unBindMsg(int id);
```

The *`regexp`* follows the PCRE syntax (see man pcrepattern(3)), grouping is done with brackets. The *`callback`* is an object implementing the IvyMessageListener interface, with the `receive` method. The thread listening on the connexion with the sending agent will execute the callback.

There are two ways of defining the callback: the first one is to make an object an implementation of the IvyMessageListener interface, and to implement the `public void receive(Ivyclient ic, String[] args)` method. But this is limited to one method per class, so the second method used is the one of inner classes, introduced since java 1.1 and widely used in swing programs, for instance:

```
bindMsg("^a*(.*)c*$", new IvyMessageListener() {
  public void receive(IvyClient ic,String[] args) {
    ... // do some stuff
  }
});
```

The processing of the ivy protocol and the execution of the callback are performed within an unique thread per remote client. Thus, the callback will be performed sequentially. If you want an asynchronous handling of callbacks, see in the advanced functions.

5.4. Subscribing to application events

TODO

6. Advanced functions

6.1. Sending to self

By default, an application doesn't send the messages to itself. Usually, there are more efficient and convenient ways to communicate withing a program. However, if you want to take benefit of the ease of ivy or to be as transparent as possible, you can set the Ivy object so that the pattern matching and message sending will be done for the sender too. This method exists since 1.2.4.

```
public void sendToSelf(boolean b);
public boolean isSendToSelf();
```

6.2. Newline within messages

As we have seen in `Ivy.sendMsg()`, you can not have newline characters within the string you send on the bus. If you still want to send messages with newline, you can encode and decode them at the emitter and receiver's side. With **Ivy.protectNewLine(boolean b)**, you can set your Ivy object to ensure encoding and decoding of newlines characters. This is tested and working between java ivy applications, but not yet implemented in other ivy libraries. The newlines are replaced by ESC characters (hex 0x1A). As the encoding and decoding cost a little more CPU and is not yet standardized in the Ivy protocol, use it at your own risk. We should of course protect the other protocol special caracters.

6.3. Sending direct messages

Direct messages is an ivy feature allowing the exchange of information between two ivy clients. It overrides the subscription mechanism, making the exchange faster (there is no regexp matching, etc). However, this features breaks the software bus metaphor, and should be replaced with the relevant bounded regexps, at the cost of a small CPU overhead. The full direct message mechanism in java has been made available since the ivy-java-1.2.3, but i won't document it to make it harder to use.

6.4. Asynchronous Subscription to messages

For each and every remote agent on the bus, a thread is in charge of handling the encoding and decoding of the messages and of the execution of the callbacks. Thus, if a callback consumes much time, the rest of the communication is put on hold and the processing is serialized, eventually leading to a stacking in the socket buffer and to the blocking of the message sender. To alleviate this, we have set up since ivy-java 1.2.4 an asynchronous subscription, where each and every time a callback is performed, it is done in a newly created separate thread. As creating a thread is quite expensive, one should use this method for lengthy callbacks only. Furthermore, to avoid concurrent access to the callback data, the String[] argument passed on to the callbacks are cloned. This causes an extra overhead.

```
public int bindMsg(String regexp, IvyMessageListener callback,boolean async);
public int bindAsyncMsg(String regexp, IvyMessageListener callback);
```

If the *`async`* boolean parameter is set to true, a new thread will be created for each callback. The same `unBindMsg()` can be called to cancel a subscription.

6.5. Waiting for someone: waitForClient and waitForMsg

Very often, while developping an Ivy agent, you will be facing the need of the arrival of another agent on the bus to perform your task correctly. For instance, for your spiffy application to run, a gesture recognition engine will have to be on the bus, or another data sending application. The Ivy way to do this is to subscribe to the known agent's *`ready message`* (be sure to subscribe before starting the bus), or to implement an

IvyApplicationListener and change of state in the `connect()` method. However, it is often useful to stop and wait, and it is awkward to wait for a variable change.

```
IvyClient waitForClient(String name, int timeout)
IvyClient waitForMsg(String regexp, int timeout)
```

These two methods allow you to stop the flow of your main (or other) thread by waiting for the arrival of an agent, or for the arrival of a message. If the agent is already here, `waitForClient` will return immediately. If *`timeout`* is set to null, your thread can wait "forever", otherwise it will wait *`timeout`* milliseconds. With `waitForMsg`, be aware that your subscription can be propagated to the remote agents after that their message was sent, so that you'd wait for nothing. You had better be sure that the `waitForMsg` method is called early enough.

6.6. Subscribing to subscriptions

TODO

7. Utilities

7.1. Probe

Probe is your swiss army knife as an Ivy java developper. Use it to try your regular expressions, to check the installation of the system, to log the messages, etc. To use it, either run fr.dgac.ivy.Probe, or run the jar file directly with **$ java -jar ivy.jar**

The command line options (available with the -h command line switch) are the following:

- -b allows you to specify the ivy bus. This overrides the -DIVY_BUS java property. The default value is 127.255.255.255:2010.
- -n NAME allows you to specify the name of this probe agent on the bus. It defaults to JPROBE, but it might be difficult to differenciate which jprobe sent which message with a handful of agents with the same name
- -q allows you to spawn a silent jprobe, with no terminal output
- -s sends to self (default off), allows subscription to its own messages
- -n NEWNAME changes JPROBE default Ivy name to another one, which can prove to be useful when running different probes
- -t add timestamps to messages
- -d allows you to use JPROBE on debug mode. It is the same as setting the VY_DEBUG property (java -DIVY_DEBUG fr.dgac.ivy.Probe is the same as java fr.dgac.ivy.Probe -d)
- -h dumps the command line options help.

The run time commands are preceded by a single dot (.) at the beginning of the line. Issue ".help" at the prompt (without the double quotes) to have the list of availables comands. If the lines does not begin with a dot, jprobe tries to send the message to the other agents, if their subscriptions allows it. The dot commands are the following

- .die CLIENTNAME issues an ivy die command, presumably forcing the first agent with this name to leave the bus
- .bye (or .quit) forces the JPROBE application to exit. This is the same as issing an end of file character on a single input line (^D).
- .direct client id message sends the direct message to the remote client, using the numeric id
- .bind REGEXP and .unbind REGEXP will change Probe's subscription
- .list gives the list of clients seen on the ivy bus

7.2. IvyDaemon

As the launching and quitting of an ivy bus is a bit slow, it is not convenient to spawn an Ivy client each time we want to send a simple message. To do so, we can use the IvyDaemon, which is a TCP daemon sitting and waiting on the port 3456, and also connected on the default bus. Each time a remote application connects to this port, every line read until EOF will be forwarded on the bus. The standard port and bus domain can be overriden by command line switches (use **$ java fr.dgac.ivy.IvyDaemon -h**). First, spawn an ivy Damon: **$ java fr.dgac.ivy.IvyDaemon** then, within your shell scripts, use a short tcp connexion (for instance netcat): **$ echo "hello world" | nc -q 0 localhost 3456** The "hello world" message will be sent on the default Ivy Bus to anyone having subscribe to a matching pattern

8. programmer's style guide

TODO

9. Contacting the author

The Ivy java library is now maintained by Yannick Jestin. For bug reports or comments on the library itself or about this document, please send him an email at <`jestin@cena.fr`>. For comments and ideas about Ivy itself (protocol, applications, etc), please join and use the Ivy mailing list <`ivy@cena.fr`>.