

---

# The Ivy Java library guide

CENA NT02-819

Yannick Jestin <yannick.jestin@enac.fr>

Copyright © 2006 DGAC/DSNA/DTI

## Abstract

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 1.2.12 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/>].

## Table of Contents

Foreword .....	2
What is Ivy? .....	2
The Ivy Java library .....	2
What is it? .....	2
Getting and installing the Ivy Java library .....	2
Your first Ivy application .....	3
The code .....	3
Compiling it .....	4
Testing .....	4
Basic functions .....	5
Initialization an Ivy object and joining the bus .....	5
Emitting messages .....	6
Subscription to messages .....	6
Subscribing to application events .....	7
Advanced functions .....	7
Sending to self .....	7
Initializing a domain .....	7
Newline within messages .....	8
Sending direct messages .....	8
Ivy and swing GUI .....	8
Asynchronous Subscription to messages .....	8
Waiting for someone: waitForClient and waitForMsg .....	8
Subscribing to subscriptions .....	9
Monitoring the bus .....	9
Message classes .....	9
Utilities .....	10
jprobe .....	10
IvyDaemon .....	11
jafter .....	11
programmer's style guide .....	11
join the right bus .....	11
nice regular expressions .....	12
Something's not working .....	12
time consuming callbacks .....	12
how to perform requests .....	12

how to quit the application ? .....	13
Contacting the authors .....	14

## 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.

## What is Ivy?

Ivy is a software bus initially 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 libraries 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 software 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/>] .

## The Ivy Java library

### 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 connect any thread-safe Java application to an Ivy bus. So far, this library has been tested and used on a variety of Java virtual machines (from 1.1.7 to 1.5.0), a variety of vendors (kaffe+gcj, sun jdk, blackdown) and on a variety of architectures (GNU/Linux, Solaris, Windows NT,XP,2000, MacOSX). It is developed and maintain on a Debian GNU/Linux unstable distribution.

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

### Getting and installing the Ivy Java library

You can get the latest versions of the Ivy C library from the Ivy web site . 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 small set of tools ( IvyDaemon, jrobe, jafter). 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.tools.Probe
```

It 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. Alternatively, you can use the jprobe shell script.

```
$ jprobe
```

## Your first Ivy application

We are going to write a "Hello world translator" 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.

### The code

Here is the code of "ivyTranslator.java":

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

class ivyTranslator implements IvyMessageListener {

    private Ivy bus;

    ivyTranslator() throws IvyException {
        // initialization, name and ready message
        bus = new Ivy("IvyTranslator","IvyTranslator 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.sendMessage("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 ivyTranslator();
}
}
```

## Compiling it

You should be able to compile the application with the following command (the classpath may vary):

```
$ javac -cp /usr/share/java/ivy.jar ivyTranslator.java
```

## Testing

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

```
$ java cp ./usr/share/java/regexp.jar:/usr/share/java/ivy.jar ivyTranslator
```

In another shell, launch **java fr.dgac.ivy.tools.Probe '(\*)'** . You can see that the IvyTranslator 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.tools.Probe '(*)'

you want to subscribe to (.* )
broadcasting on 127.255.255.255:2010
IvyTranslator connected
IvyTranslator subscribes to ^Bye$
IvyTranslator subscribes to ^Hello(.* )
IvyTranslator sent 'IvyTranslator Ready'
```

Probe is an interactive program. Type "Hello Paul", and you should receive "Bonjour Paul". Type "Bye", and the ivyTranslator 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
```

```
IvyTranslator sent 'Bonjour Paul'
```

```
Bye
```

```
-> Sent to 1 peers
```

```
IvyTranslator disconnected
```

```
<Ctrl-D>
```

```
$
```

## 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.

## 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 regexprs, direct messages or die command issued by other agents.

At this point, your 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 eventually 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 be 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 protocol* 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 stoppped with `Ivy.stop()` or when a die message is received. The rendezvous 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.

## 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.

## 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 the pcre web site [<http://www.pcre.org/>] or the `pcrepattern(3)` man page), 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 anonymous 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.

## Subscribing to application events

Either at the creation time of your Ivy object or later on with `Ivy.addApplicationListener()`, you can add some behaviour to perform callbacks upon different events:

- when an agent joins the bus
- when an agent leaves the bus
- when an agent sends you a direct message
- when an agent forces you to leave the bus

This can be handy if you design an agent requiring some coordination with other dedicated agents to run properly. To do so, the easiest way is to use the ready messages. You can find later a description of the `Ivy.waitForClient()` method to implement a correct synchronization.

## Advanced functions

### 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.

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

### Initializing a domain

The default behaviour of an Ivy agent is to accept a command line switch (`-b 10:2010`, e.g. ), and if not present, to use the `IVYBUS` property, ( given by the `-DIVYBUS=10:34567` parameter to the jvm ), and, if not present, to default to `Ivy.DEFAULT_DOMAIN`. This domain is given as a string arduement to the `Ivy.start()` function. To make this logic easier to follow, the Ivy class provides the programmer with two useful function:

```
public static String getDomain(String arg);
public static String getDomainArgs(String progname,String[] args);
```

The `Ivy.getDomain()` function, if `arg` is non null, will return `arg`, otherwise it will return the `IVY-BUS` property, otherwise the `DEFAULT_DOMAIN`. A very simple way to start an Ivy agent is with

`Ivy.start(getDomain(null))`. The `getDomainArgs(name,args)` will add very simple processing of the args given to the `main()` function, and give higher priority to the command line argument.

## Newline within messages

As we have seen in `Ivy.sendMessage()`, 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 standardised in the Ivy protocol, use it at your own risk. We should of course protect the other protocol special characters.

## 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 it won't be much documented, in order to make it harder to use.

## Ivy and swing GUI

Swing requires the code to run in the main swing thread. In order to avoid problems, be sure tu use the `SwingUtilities.invokeLater()` or `SwingUtilities.invokeAndWait()` methods if you Ivy callbacks impact swing components.

## 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 serialised, 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 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.

## Waiting for someone: waitForClient and waitForMsg

Very often, while developing 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.

## Subscribing to subscriptions

A very common practice when beginning to play with ivy is to develop an ivy agent monitor (the good practice is to use the excellent `ivymon` written in perl by Daniel Etienne). If you want to notify the user that a remote agent has subscribed or unsubscribed to a regular expression after the protocol handshake, then your monitor agent has to subscribe to subscriptions. To do so, use the following functions:

```
public int addBindListener(IvyBindListener callback);
public void removeBindListener(int id)
```

A `IvyBindListener` object must implement the following interface:

```
void bindPerformed(IvyClient client, int id, String regexp);
void unbindPerformed(IvyClient client, int id, String regexp);
```

For a code sample, see the `Probe` utility source code. Note that if you have enabled a filter (message classes), you will be notified the subscriptions even if they are considered useless. If you want to check if the regexp has a chance to match the message you're sending, use the boolean `Ivy.CheckRegexp(String regexp)`.

## Monitoring the bus

When in doubt, to check if the remote client is still responding, instead of relying on a callback, you can test the response of the protocol parsing thread. Use following method:

```
Ivyclient.ping(PingCallback pc);
```

It will send a ping token that will (hopefully) be parsed by the remote agent, and will trigger the following message with the elapsed time in milliseconds:

```
void PingCallback.pongReceived(IvyClient ic,int elapsedTime);
```

An example is provided in `fr.dgac.ivy.tools.Probe`.

## Message classes

When your Ivy bus is populated with many agents, the cost of pattern matching becomes painful. For instance, a bus with 20+ agents, with 2000+ subscriptions, with hundreds of messages per second might cause a high CPU load, thus leading to slow responsiveness in GUI animations. To limit this phenomenon,

use bounded regexp as much as possible like `^NAME VALUE= ( . * )` ( see the programmer's style guide later on ). However, 2000+ subscription are still 2000+ tests of bounded regexp, even if is less costful. It is possible not to do the tests provided some requirements are met:

- your agent knows exhaustively and exactly the prefixes of all the messages it will send
- you use the `Ivy.setFilter()` before starting the bus, with the list of prefixes ( e.g.: { "TOTO1" "PREFIX", "ETC" } )

If you use the message classes, your Ivy agent will ignore the bounded subscriptions of other agents that will never match any of your prefixes ( e.g: `^COUCOU` neither matches `TOTO1`, `PREFIX`, nor `ETC` ). The check is made this way:

- if the regexp is bounded, the filter extracts the first word according to this regexp: `^\^\^([a-zA-Z0-9_-]+) . *`
- the word is compared character to character to all the prefixes in the message class; if it is not present, the subscription is discarded

When your agent sends a message, many pattern matching won't be made, and it might save some time. Be sure to activate the `setFilter()` when you are sure that you know perfectly the message classes. You can play with the message classes with `jprobe` and see the problems that can arise.

## Utilities

### jprobe

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

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 differentiate 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.tools.Probe` is the same as `java fr.dgac.ivy.tools.Probe -d` )
- `-c MESSAGECLASS` uses a message filter (see `Ivy.setFilter()` ), for example `'Word1,Word2,Word3'`
- `-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 available commands. 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 inputting 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
- .bound AGENT lists the regexps the AGENT has subscribed to. You can use .bound \* to get the whole list.
- .time COUNT MSG sends the MSG COUNT times and displays the elapsed time
- .ping CLIENT measures the time it takes to reach a client

## 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 overridden by command line switches ( use **\$ java fr.dgac.ivy.tools.IvyDaemon -h** ). First, spawn an ivy Damon: **\$ java fr.dgac.ivy.tools.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

## jafter

jafter ( or fr.dgac.ivy.tools.JAfter ) is a simple utility that can be used within shell script. The rationale is to block the processing until a specific message is received, then continue. This can be used to wait for a ready message before launching another agent. The jafter program can wait forever or timeout. If the timeout is triggered, a negative value is returned to the shell.

## programmer's style guide

There are many specific programming patterns in Ivy. Some of them are Ivy related, some are java related. See the jprobe source code to get an example of some programming patterns. Here are some of them, to be completed later...

## join the right bus

To join the right bus, you have to honor the IVYBUS property. It is a good way to let the system get it, and an ever better way to override it with command line options (e.g.: -b :3110). Here is a snippet to perform this task:

```
import fr.dgac.ivy.* ;
import gnu.getopt.Getopt;

public static void main(String[] args) throws IvyException {
    Getopt opt = new Getopt("After",args,"b:"); // add more options ...
    String domain=Ivy.getDomain(null); // gets IVYDOMAIN from property
    // or, if none is set, defaults to Ivy.DEFAULT_DOMAIN
    while ((c = opt.getopt()) != -1) switch (c) {
        case 'b': domain=opt.getOptarg(); break; // overrides
        // and more options
        default: System.out.println(helpmsg); System.exit(0);
    }
    Ivy bus=new Ivy(name,name+" ready",null);
    bus.start(domain); // sets the properties for children processes ...
}
```

## nice regular expressions

To avoid CPU consuming pattern matching operation, be sure to use bounded regexps as much as possible. For instance, if we consider the regexp1 "^coord x=(d+)", the regexp2 "x=(d+)" and the message msg "coord x=12 y=15". When another agent sends the message msg, both regexp will match and trigger the callback with an argument, the string "12". However, when another message is sent, the regexp1 will fail as soon as possible, probably the very first character, but the regexp2 will do some processing before failing.

## Something's not working

To trace the behaviour of an heavily multithread application is quite tedious, especially when it's connected to others. The easiest path is to use the built-in basic tracing mechanism provided by Ivy : run your jvm with the -DIVY\_DEBUG property set. Use jprobe to monitor what's going on with the greedy regexp '(.\*)' in a separate terminal. If in doubt, just join the Ivy mailing list.

## time consuming callbacks

For each remote agent, the Ivy object has an IvyClient with a dedicated working thread. This thread deals with the Ivy protocol coding and decoding, and performs the callbacks. If your agent has time consuming callbacks, involving CPU, or long IO, or so, then it might be better to run each callback in a dedicated thread. You can write this yourself, or just use the Ivy.bindAsyncMsg() function. The problem is that there is a slight overhead in thread management.

## how to perform requests

When agent A needs to make a request to another agent B, you can use the following pattern:

- B has subscribed to ^MyRequest ID=( [^ ]+ ) QUERY=(.\*)
- A subscribes to ^MyResult ID=someSpecificId RESULT=(.\*)
- A send to MyRequest ID=someSpecificId QUERY=2+2
- B receives, computes the results, then sends MyResult ID=someSpecificId RESULT=4
- A receives, and unsubscribes to his subscription

To program this, you have to get a gentle way of choosing the specificId, associate this to the subscription number returned by `Ivy.bindMsg()`, and add code within the callback to perform the unsubscription. Moreover, you have to be sure that there are not two "B type" agents on the bus, otherwise you'll eventually perform your callback twice if the results are sent during the short lapse of time before unsubscription.

The preferred way is to let the API provide this singleton mechanism with `Ivy.getWBUID()` function returning an unique ID, and the `Ivy.bindMsgOnce()` that handles all the mechanisms ensuring that the callback will be run on time only, and that the unsubscription will take place.

```
String id = bus.getWBUID(); // a more or less unique ID
bus.bindMsgOnce("MyResult ID="+id+" RESULT=(.*)",{
    public void receive(IvyClient ic,String[] args) {
        System.out.println("2+2="+args[0]);
        // the unsubscription is done for me
    }
});
bus.sendMessage("MyRequest ID="+id+" QUER=2+2");
```

## how to quit the application ?

If your application decides to quit or leave the bus, the safest way is to invoke `Ivy.stop()`. This will send a clean goodbye message to the other agents, close the sockets, end the threads, etc. However, you can en the JVM performing a `System.exit(int)` and let the other agents realize that you've gone.

If some other agents wants you to quit the bus, it will send you a die message. The protocol will first run you `IvyApplicationListener.die()`, if any, then perform some socket/thread clean up, and disconnect you from the bus. As a good citizen on the Ivy bus, you should take the appropriate measures top stop your application within the `IvyApplicationListener.die()` method, for instance run `System.exit()` or make all you thread stop, disposing your toplevel Swing Frames. Here is an example of how to do it:

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

public class EndApp extends IvyApplicationAdapter {

    public static void main(String[] args) throws IvyException {
        Ivy bus=new Ivy("EndApp","EndApp ready",null);
        EndApp e = new EndApp(bus); // a frame is opened, and the Swing Thread is star
        bus.addApplicationListener(e);
        bus.start(Ivy.getDomain(null)); // Ivy threads are up and running
        // the control flow won't stop until the end of all above threads
    }

    private Ivy bus;
    JFrame f;

    public EndApp(Ivy b) {
        this.bus=b;
        f=new JFrame("test");
        f.getContentPane().add(new JLabel("some label"),java.awt.BorderLayout.CENTER);
        f.pack();
    }
}
```

```
        f.setVisible(true);
    }

    public void die(IvyClient client, int id,String msgarg) {
        System.out.println("received die msg from " + client.getApplicationName());
        f.dispose(); // closes the only window, thus quitting the swing thread
    } // end of die callback, the Ivy threads are stopped

}
```

## Contacting the authors

For bug reports or comments on the library itself or about this document, please me an email at <yannick.jestin@enac.fr>. For comments and ideas about Ivy itself (protocol, applications, etc), please join and use the Ivy mailing list [<http://www.tls.cena.fr/products/ivy/contact.html>]

If you report a bug, try to identify the causal path leading to the bug, and submit a trace of the problem, if possible, using the `-DIVY_DEBUG` property to produce a trace of the ivy execution.