The Ivy C library guide

Francois-Régis Colin

fcolin@cena.fr

Stéphane Chatty

chatty@cena.fr

Yannick Jestin

jestin@cena.fr

This document is a programmer's guide that describes how to use the Ivy C library to connect applications to an Ivy bus. This guide describes version 3.9 of the library.

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 so ugly. We'll try and improve it.

2. What is Ivy?

Ivy is a software bus designed at CENA (France). 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 C 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*. All those documents should be available from the Ivy Web site at http://www.tls.cena.fr/products/ivy/.

3. The Ivy C library

3.1. What is it?

The Ivy C library (aka Ivy-C or ivy-c) is a C library that allows you to connect applications to an Ivy bus. You can use it to write applications in C or any other language that supports C extensions. You can also use it to integrate an application that already has a main loop (such as a GUI application) within an Ivy bus. This guide is here to help you do that.

The Ivy C library is known to compile and work in WindowsNT and Linux environments. It should be easy to use on most Posix environments.

The Ivy C library was originally developed by François-Régis Colin at CENA. It is maintained by a group at CENA (Toulouse, France)

3.2. Getting and installing the Ivy C library

You can get the latest versions of the Ivy C library from CENA (http://www.tls.cena.fr/products/ivy/). Depending on whether you use a supported binary distribution, you can retrieve binary RPM or Debian packages for Linux (do not forget to get the development package as well as the run-time package), or retrieve the source files and compile them.

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 "Hello", and re-emit them after translating "Hello" into "Bonjour". In addition,

the application will quit when it receives any message containing exactly "Bye".

4.1. The code

Here is the code of "hellotranslater.c":

```
#include <stdlib.h>
#include <stdio.h>
#include <getopt.h>
#include <ivy.h>
#include <ivyloop.h>

/* callback associated to "Hello" messages */
void HelloCallback (IvyClientPtr app, void *data, int argc, char **argv)
{
const char* arg = (argc < 1) ? "" : argv[0];
IvySendMsg ("Bonjour%s", arg);
}

/* callback associated to "Bye" messages */
void ByeCallback (IvyClientPtr app, void *data, int argc, char **argv)
{
IvyStop ();
}

main (int argc, char**argv)
{
/* handling of -b option */
const char* bus = 0;
char c;
while (c = getopt (argc, argv, "b:") != EOF) {
switch (c) {
case 'b':
bus = optarg;
break;
}
}

/* handling of environment variable */
if (!bus)
bus = getenv ("IVYBUS");

/* initializations */
IvyInit ("IvyTranslater", "Hello le monde", 0, 0, 0, 0);
IvyStart (bus);

/* binding of HelloCallback to messages starting with 'Hello' */
IvyBindMsg (HelloCallback, 0, "^Hello(.*)");

/* binding of ByeCallback to 'Bye' */
IvyBindMsg (ByeCallback, 0, "^Bye$");
```

```
/* main loop */
IvyMainLoop();
}
```

4.2. Compiling it

On a Unix computer, you should be able to compile the application with the following command:

```
$ cc -o ivytranslater ivytranslater.c -livy
$
```

4.3. Testing

We are going to test our application with **ivyprobe**. In a terminal window, launch **ivytranslater**.

```
$ ivytranslater
```

Then in another terminal window, launch **ivyprobe '(.*)'**. You are then ready to start. Type "Hello Paul", and you should get "Bonjour Paul". Type "Bye", and your application should quit:

```
$ ivyprobe '(.*)'
IvyTranslater connected from localhost
IvyTranslater subscribes to 'Hello (.*)'
IvyTranslater subscribes to 'Bye'
Hello Paul
IvyTranslater sent 'Bonjour Paul'
Bye
IvyTranslater disconnected from localhost
<Ctrl-D>
$
```

5. Basic functions

5.1. Initialization and main loop

Initializing an Ivy agent with the Ivy C library is a two step process. First of all, you should initialize the library by calling function `IvyInit`. Once the library is initialized you can create timers and add

subscriptions, but your agent is still not connected to any bus. In order to connect, you should call function `IvyStart`. In theory, initialization is then over. However in practice, as for any asynchronous communication or interaction library, nothing happens until your application has reached the main loop.

The Ivy C library provides its own main loop: `IvyMainLoop`. You should use it unless you already use a toolkit that provides its own main loop and you want to use that one. If it is the case, please refer to section XX. Otherwise, just call `IvyMainLoop`. From within the main loop, you can call `IvyStop` to exit the loop.

Here are more details on those functions:

```
void IvyInit (const char* agentname,
const char* ready_msg,
IvyApplicationCallback app_cb,
void *app_data,
IvyDieCallback die_cb,
void *die_data);
```

initializes the library. *`agentname`* is the name of your application on the Ivy bus. It will be transmitted to other applications and possibly used by them, as does **ivyprobe**. *`ready_msg`* is the first message that is going to be sent to peer applications, bypassing the normal broadcasting scheme of Ivy (see *The Ivy architecture and protocol* for more details). If a zero value is passed, no message will be sent. *`app_cb`* is a callback that will be called every time a new peer is detected. If a zero value is passed, no callback is called. *`app_data`* is a pointer that will be passed to the application-connection callback. *`die_cb`* is a callback that will be called every time a peer disconnects. If a zero value is passed, no callback is called. *`die_data`* is a pointer that will be passed to the application-disconnection callback.

```
void IvyStart (const char* bus);
```

connects your application to the bus specified in *`bus`*. The string provided should follow the convention described in section XX. Example: *`"10.192.33,10.192.34:2345"`*. If a null value is passed, the library will use the value of the environment variable *`IVYBUS`*, which should have the same syntax. If the environment variable is not defined, the default value *`"127:2010"`* is used.

```
void IvyMainLoop ();
```

makes your application enter the main loop in which it will handle asynchronous communications and signals. This is the default Ivy main loop, based on the `select` POSIX system call. You can interact with the mainloop using hook before and after `select` use the `IvySetBeforeSelectHook` and `IvySetAfterSelectHook`

```
void IvyStop ();
```

makes your application exit the main loop.

5.2. Emitting messages

Emitting a message on an Ivy bus is much like printing a message on the standard output. However, do not forget that your message will not be emitted if Ivy has not been properly initialized and if you do not have a main loop of some sort running. To emit a message, use `IvySendMsg`, which works like `printf`:

```
void IvySendMsg (const char* format, ...);
```

sends a message on the bus. This function has exactly the same behaviour as `printf`, `sprintf` or `fprintf`.

5.3. Subscribing to messages

Subscribing to messages consists in binding a callback function to a message pattern. Patterns are described by regular expressions with captures. When a message matching the regular expression is detected on the bus, the 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 function `IvyBindMsg` to bind a callback to a pattern, and function `IvyUnbindMsg` to delete the binding.

```
MsgRcvPtr IvyBindMsg (MsgCallback cb,
void* data,
const char* regex_format, ...);
```

binds callback function `cb` to the regular expression specified by *`regex_format`* and the optional following arguments. *`regex_format`* and the following arguments are handled as in `printf`. The return value is an identifier that can be used later for cancelling the subscription. There is a special syntax for specifying numeric interval, in this case the interval is locally transformed in a pcre regexp. syntax is (?Imin#max[fi]). min and max are the bounds, by default the regexp match decimal number, but if max bound is followed by 'i', the regexp match only integers ex : (?I-10#20), (?I20#25i) Note that due to the regexp generator limitation (which will perhaps be raised eventually) the bounds are always integers.

```
void IvyUnbindMsg (MsgRcvPtr id);
```

deletes the binding specified by *`id`*.

In what precedes, MsgRcvPtr is an opaque type used to identify bindings, *`data`* is a user pointer passed to the callback whenever it is called, and *`MsgCallback`* is defined as follows:

```
typedef void (*MsgCallback)(IvyClientPtr app, void *data, int argc, char **argv);
```

6. Advanced functions

6.1. Utilities

[to be written]

6.2. Direct messages

[to be written]

6.3. Managing timers and other channels

In your applications, you may need to manage other input/output channels than an Ivy bus: a serial driver, the channels defined by a graphical toolkit, or simply stdin and stdout. The same applies for timers. You can either manage those channels or timers from the Ivy main loop, or instead use the main loop provided by another library.

6.3.1. Channels

You can get a channel to be managed from the Ivy main loop by using functions `IvyChannelAdd` and `IvyChannelRemove`.

```
Channel IvyChannelAdd (HANDLE fd,
void* data,
ChannelHandleDelete handle_delete,
ChannelHandleRead handle_read);
```

ensures that function `handle_read` is called whenever data is read on file descriptor *`fd`*, and function *`handle_delete`* whenever *`fd`* is closed, and

```
void IvyChannelRemove (Channel ch);
```

terminates the management of channel *`ch`*.

In what precedes, Channel is an opaque type defined by the Ivy C library, *`data`* is a pointer that will be passed to functions `handle_read` and `handle_delete`. It can be defined at will by users. The types HANDLE, ChannelHandleDelete and ChannelHandleRead are as follows:

```
typedef int HANDLE; (for Unix)
typedef SOCKET HANDLE; (for Windows)
typedef void (*ChannelHandleDelete)(void *data);
typedef void (*ChannelHandleRead)(Channel ch, HANDLE fd, void* data);
```

6.3.2. Adding timers

You can get a function to be repeatedly called by using function `TimerRepeatAfter`:

```
TimerId TimerRepeatAfter (int nbticks, long delay, TimerCb handle_timer, void* data);
```

ensures that function *`handle_timer`* is called *`nbticks`* times at intervals of *`delay`* seconds, thus creating a timer.

```
void TimerModify (TimerId id, long delay);
```

changes the delay used for timer *`id`*.

```
void TimerRemove (TimerId id);
```

deletes timer *`id`*, thus stopping it.

In what precedes, *`data`* is passed to *`handle_timer`* every time it is called. *`delay`* is expressed in milliseconds. If *`nbticks`* is set to *`TIMER_LOOP`*, then *`handle_timer`* will be called forever. TimerCb is as follows:

```
typedef void (*TimerCb)(TimerId id, void *data, unsigned long delta);
```

7. Conventions for writing applications

In addition to the Ivy protocol, Ivy applications should respect two conventions when used in a Posix environment:

- They should accept the option `-b` or `-bus` to specify the Ivy bus on which they will connect. The Ivy C library provides no support for that.
- They should refer to the environment variable *`IVYBUS`* when the above option is not used. With the Ivy C library, this is obtained by passing a null value to `IvyStart`

8. Using Ivy with another main loop

The ivyprobe source code holds examples of use of Ivy within other main loops, namely Xt and Gtk.

8.1. Using Ivy with the X Toolkit

The basics for using the Ivy withing the XtAppMainLoop() are the following ones:

- include the ivy.h and ivyxtloop.h
- link with libxtivy.o (add the `-lxtivy` ld flag and NOT the `-livy`)
- create the ivy bus
  - IvyXtChannelAppContect(app_context) with an existing Xt context
  - You can add channels to be handled by Ivy, for instance, stdin, with the IvyChannelAdd function
  - IvyInit(char *name,char *readyMessage,IvyApplicationCallback cb,void *cbUserData,IvyDieCallback dieCb,void *dieCbUserdata)
  - IvyBindMsg() for the behavior
  - IvyStart(char *domain)
- run the Xt main loop with XtAppMainLoop(app_context)

Here is an example, motifButtonIvy.c. You can compile it with the following command line:

```
cc -o motifButtonIvy motifButtonIvy.c -lxtivy
```

The result is a simple single-buttoned application emitting a message on the bus. The message defaults to "foo", but can be updated via an Ivy Button text=bar message.

```
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <Xm/PushB.h>
#include <ivy.h>
#include <ivyxtloop.h>

void myMotifCallback(Widget w,XtPointer client_d,XtPointer call_d){
  IvySendMsg (*((char**)client_d));
}
void textCallback(IvyClientPtr app, void *user_data, int argc, char *argv[]){
  *((char **)user_data)=argv[0];
}
void DieCallback (IvyClientPtr app, void *data, int id){
  exit(0);
}
int main(int argc,char *argv[]){
  Widget toplevel,pushb;
  XtAppContext app_context;
  Arg myargs[10];
  char *bus=getenv("IVYBUS");
  char *tosend="foo";
  toplevel=XtAppInitialize(&app_context,"Ivy Button",NULL,0,&argc,argv,NULL,myargs,0);
  pushb=XmCreatePushButton(toplevel,"send message",myargs,1);
```

```
  XtManageChild(pushb);
  XtAddCallback(pushb,XmNactivateCallback,myMotifCallback,&tosend);
  XtRealizeWidget(toplevel);
  IvyXtChannelAppContext(app_context);
  IvyInit("IvyMotif","IvyMotif connected",NULL,NULL,DieCallback,NULL);
  IvyBindMsg(textCallback,&tosend,"^Ivy Button text=(.*)");
  IvyStart(bus);
  XtAppMainLoop(app_context);
}
```

8.2. Using Ivy with Tcl/Tk

Just load the libtclivy.so package, and use the following commands

```
#!/usr/bin/tclsh
Ivy::init $name $hellomessge connectproc dieproc
Ivy::start $domain
Ivy::bind $regexp ballback
Ivy::applist
Ivy::send $message
Ivy::applist
mainloop
```

A full example in Tcl/Tk is provided here:

```
#!/usr/bin/wish
load libtclivy.so.3.4
proc connect {args} { }
proc send { } {
  global tosend
  Ivy::send $tosend
}
proc dotext {text} {
  global tosend
  set tosend $text
}
Ivy::init "IvyTCLTK" "IvyTCLTK READY" connect echo
Ivy::start 127.255.255.255:2010
Ivy::bind "^Ivy Button text=(.*)" dotext
set tosend foo
button .send -command send -text "send msg"
pack .send
```

8.3. Using Ivy with Gtk

There is little to do to make your gtk applications Ivy aware: just add the following lines into your code:

```
#include <ivy.h>
#include <ivygtkloop.h>
...
IvyInit ("IvyGtkButton", "IvyGtkButton READY",NULL,NULL,NULL,NULL);
IvyBindMsg(textCallback,&tosend,"^Ivy Button text=(.*)");
IvyStart (bus);
```

A full example: gtkIvyButton.c is provided below, compile it with the -lgtkivy flag. The other flags depend on your system installation (replace pkg-config with gtk-config for older gnome1 libs)

```
#include <gtk/gtk.h>
#include <ivy.h>
#include <ivygtkloop.h>
#include <stdio.h>
#include <stdlib.h>

void hello( GtkWidget *widget, gpointer   data ) {
  fprintf(stderr,"%s\n",*((char**)data));
  IvySendMsg(*((char**)data));
}

void textCallback(IvyClientPtr app, void *user_data, int argc, char *argv[]){
  *((char **)user_data)=argv[0];
}

int main( int   argc, char *argv[] ) {
    GtkWidget *window;
    GtkWidget *button;
    char *bus=getenv("IVYBUS");
    char *tosend="foo";
    gtk_init (&argc, &argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    button = gtk_button_new_with_label ("send message");
    g_signal_connect (G_OBJECT(button),"clicked",G_CALLBACK(hello),&tosend);
    gtk_container_add (GTK_CONTAINER(window),button);
    gtk_widget_show (button);
    gtk_widget_show (window);
    IvyInit ("IvyGtkButton", "IvyGtkButton READY",NULL,NULL,NULL,NULL);
    IvyBindMsg(textCallback,&tosend,"^Ivy Button text=(.*)");
    IvyStart (bus);
    gtk_main ();
    return 0;
}
```

8.4. Adding Ivy to another main loop

8.4.1. Functions to be provided

You can decide to use the main loop from another toolkit than the X Toolkit or the Tk toolkit. If you do that, you'll have to define four functions that Ivy will use to get its own channels managed by the other toolkit. you should link ivy with your new module insted of the ivy(xxx)loop module. These functions are declared in ivychannel.h:

`IvyChannelInit`

`IvyChannelStop`

`IvyChannelAdd`
`IvyChannelRemove`

They should point to functions that respectively:

- make the necessary global initializations before entering the main loop
- make the necessary global finalizations before exiting the main loop
- initialize a channel and ensure that it is managed by the main loop
- close a channel

The types ChannelInit, ChannelSetUp and ChannelClose are defined as follows:

```
extern void IvyChannelInit(void);


extern void IvyChannelStop (void);


/* function called by Ivy to set callback on the sockets */

extern Channel IvyChannelAdd(

        HANDLE fd,

        void *data,

        ChannelHandleDelete handle_delete,

        ChannelHandleRead handle_read
```

```
);


/* function called by Ivy to remove callback on the sockets */

extern void IvyChannelRemove( Channel channel );
```

8.4.2. Type to be defined

In order to implement the three previous functions, you will need to define the hidden type struct _channel (the type Channel is defined as struct _channel*). Use it to store the data provided by the other toolkit.

9. Contacting the authors

The Ivy C library was mainly written by Francois-Régis Colin, with support from Stéphane Chatty. For bug reports or comments on the library itself or about this document, please send them an email: fcolin@cena.fr and chatty@cena.fr. For comments and ideas about Ivy itself (protocol, applications, etc), please use the Ivy mailing list: ivy@tls.cena.fr.