Jive Application (JIVE)

version 1.3

Kaj Nygren, Joakim Grebenö

1997-05-02

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

Chapter 1

JIVE User's Guide

The Jive Application (*JIVE*) is the glue which allows Java Applets/Applications to interact with Erlang.

Java and Erlang have Jive packages, which hide the socket communication from the programmer. The Java side also contains a number of wrapper classes for each Erlang variable type.

Java Applets/Applications can interact with Erlang; that is,

- Spawn new Erlang processes.
- Send messages to and from Erlang processes.
- Do `apply/3` on Erlang functions.

1.1 Jive

Jive is an Erlang Application which makes it possible for a Java Applet/Application to communicate with an Erlang server. Java is ideal for client-side interaction, whereas Erlang is ideal for server-side programming. The idea behind Jive is to integrate these two languages. Jive allows a Java Applet/Application to interact with an Erlang server.

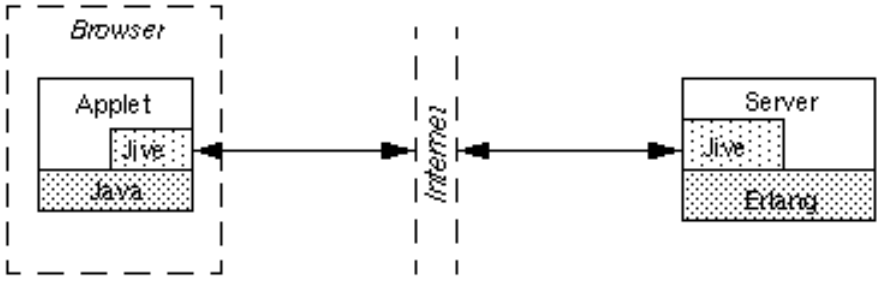

Figure 1.1: The Jive Architecture

Communication between the client and the server is socket based and both Java and Erlang has Jive packages which hide the socket communication from the programmer. Furthermore, the Java side contains a number of wrapper classes for each of the Erlang variable types (See “Concurrent Programming in ERLANG”/ISBN 0-13-508301-X).

Java -> Erlang

A Java client can interact with Erlang using three mechanisms:

- spawn new Erlang processes
- do Erlang `apply` on functions
- send messages to Erlang processes.

Erlang -> Java

Erlang processes can send messages to Java objects.

1.2 An Example

This section demonstrates how Jive is used to communicate between Erlang and a Java client, such as a Java Applet or a Java Application.

The example shown consists of a Java client which displays a clock. The Java client spawns a clock process on the Erlang server which sends a time stamp back to the Java client every second.

In the case of an Applet, an Erlang process is automatically spawned when a user visits a HTML page which has the appropriate <APPLET> tags. The Erlang process is killed automatically when the Applet is stopped.

In the case of an Application, an Erlang process is automatically spawned when a user starts the appropriate Java Application. The Erlang process is killed automatically when the Java Application is stopped.

This example together with *detailed* instructions can be found in the Jive distribution package (`$ERLANG_ROOT/lib/jive-2.x/examples/`).

Do the following to test the Java Applet and the Java Application examples:

1. start the Erlang server (`clock.erl`) on a host of your choice; Example: `clock:init()`.
2. start an appletviewer of your choice and access the HTML file (`timer.html`); Example: `appletviewer timer.html`
3. start a Java Application (TimerAlt.java); Example: `java TimerAlt akvavit 14000`.

> Warning:
> If the Java Applet is put in a Web-server the Erlang server needs to be started on the *same* host. This is true if the Web browser used only supports “Applet Host” security mode; Example: Netscape Navigator etc.

The Java Applet (Timer.java)

```
import java.applet.Applet;
import java.awt.*;
import jive.erlang.*;

public class Timer extends EApplet implements EReceive {
  private int time=0;

  /*
   * The start method initialize the Erlang connection and spawns an
   * Erlang clock server.
   */
  public void start() {
    // Call the EApplet constructor to initialize the connection
    super.start();
```

```
    try {
      // Register this Applet so Erlang can send messages to it
      EInteger id=receiver().register(this);
      // Spawn a clock server on the Erlang node
      ESock sender=getESock();
      sender.spawn("clock","start",new EList(id,self()));
    } catch (JiveException e) {
      System.err.println(e);
    }
  }

  public void paint(Graphics g) {
    g.drawString("Timer has been running for "+time+" seconds",10,10);
  }

  public void receive(EVar var) {
    if(var.type() == EVar.EINTEGER) {
      time=((EInteger)var).value();
      repaint();
    }
  }
}
```

The HTML Page (timer.html)

This is the HTML page, which loads the Applet.

> Note:
> The PARAM tag specifies which port to use when connecting to the Erlang server.

```
<HTML>
<HEAD>
<TITLE>Jive Example</TITLE>
</HEAD>
<BODY>
<P>This example shows how a simple Java Applet communicates with Erlang
using Jive.
<P><APPLET CODE="Timer.class" HEIGHT=50 WIDTH=300>
<PARAM NAME=PORT VALUE=14000>
Sorry! Your browser cannot execute Java programs...
</APPLET>
</BODY>
</HTML>
```

The Java Application (TimerAlt.java)

```
import java.awt.*;
import jive.erlang.*;
public class TimerAlt extends EApplication implements EReceive {
  private int time=0;

  /*
   * Usage: Timer host port
   */
  public static void main(String args[]) {
    if(args.length != 2) {
      System.err.println("Usage: Timer host port");
    } else {
      try {
        int port=Integer.parseInt(args[1]);
        new TimerAlt(args[0],port);
      } catch (JiveException e) {
        System.err.println(e);
        System.exit(0);
      }
    }
  }

  /*
   * Constructor which initialize the Erlang connection and spawns an
   * Erlang clock server.
   */
  public TimerAlt(String host,int port) throws JiveException {
    // Call the EApplication constructor to initialize the connection
    super(host,port);
    // Connect to the remote server
    connect();
    // Register this object so Erlang can send messages to it
    EInteger id=receiver().register(this);
    // Spawn a clock server on the Erlang node
    ESock sender=getESock();
    sender.spawn("clock","start",new EList(id,self()));
  }
  /*
   * receive gets called each time an Erlang message is sent to an object
   * instance of this class. This function need to be implemented hence
   * the class implements the EReceive interface.
   */
  public void receive(EVar var) {
    // Print the 10 first received time stamps on stdout
    if(var.type() == EVar.EINTEGER) {
      time=((EInteger)var).value();
      if(time == 10) {
        // Disconnect from the Erlang server
        disconnect();
        System.exit(0);
      } else {
```

```
        System.out.println("Timer has been running for "+time+" seconds");
      }
    }
  }
}
```

The Erlang Server (clock.erl)

```
-module(clock).
-export([init/0, start/2]).
init() ->
  jive:start(),
  jive:allow({clock,start,2}).
start(Receiver,PidId) ->
  Pid = jive:get_pid(PidId),
  link(Pid),
  loop(Receiver,Pid, 0).
loop(Receiver,Pid,Time) ->
  receive
    after 1000 ->
      Pid ! {send, Receiver, Time}
  end,
  loop(Receiver,Pid,Time+1).
```

1.3 The Java Interface

This section describes the Java interface to Jive and includes the following topics:

- Erlang variable types
- EApplet
- EApplication
- The runtime environment.

Erlang Variable Types

Jive provides wrapper classes for each defined Erlang variable type: `integer`, `float`, `string`, `list`, `tuple`, `atom`, and `pid`. The Java Jive API provides a detailed description of the Java Interface.

EVar

EVar is an abstract class that is a super class of all other Erlang wrapper classes. Most notably, this class contains a method `type()` which returns the type of the variable.

EInteger

The EInteger class is a wrapper for an Erlang integer. The EInteger has the type `EINTEGER`.

To create a new EInteger:

```
EInteger tmp = new EInteger(int value);
```

To retrieve the integer value:

```
int val = tmp.value();
```

EFloat

The EFloat class is a wrapper for an Erlang float. The EFloat has the type `EFLOAT`.

To create a new EFloat:

```
EFloat tmp = new EFloat(double value);
```

To retrieve the double value:

```
double val = tmp.value();
```

EString

The EString class is a wrapper for an Erlang string. The EString has the type ESTRING.
To create a new EString:

```
EString tmp = new EString(String string);
```

To retrieve the String value:

```
String string = tmp.value();
```

To retrieve the length of the String:

```
int length = tmp.length();
```

EAtom

The EAtom class is a wrapper for an Erlang atom. The EAtom has the type EATOM.
To create a new EAtom:

```
EAtom tmp = new EAtom(String val);
```

To retrieve the Atom value:

```
String val = tmp.value();
```

To retrieve the length of the Atom:

```
int length = tmp.length();
```

EBinary

The EBinary class is a wrapper for an Erlang binary. The EBinary has the type EBINARY.
To create a new EBinary:

```
EBinary tmp = new EBinary(byte data[]);
```

To retrieve the byte array:

```
byte data[] = tmp.value();
```

To retrieve the length of the Binary:

```
int length = tmp.length();
```

EList

The EList class is a wrapper for an Erlang list. The EList has the type ELIST.

To create a new EList:

```
EList tmp = new EList();
// to
EList tmp = new EList(EVar var1, EVar var2, EVar var3,
                      EVar var4, EVar var5, EVar var6);
```

To create a new EList from an array of EVar:

```
List tmp = new EList(EVar vars[]);
```

To retrieve the array of EVars:

```
EVar vars[] = tmp.value();
```

To retrieve the length of the List:

```
int length = tmp.length();
```

To retrieve an enumeration of the EVars in the List:

```
EVar evar[] = tmp.elements();
```

To retrieve an EVar at a specified index:

```
EVar evar = tmp.elementAt(1);
```

ETuple

The ETuple class is a wrapper for an Erlang tuple. The ETuple has the type ETUPLE.

To create a new ETuple:

```
ETuple tmp = new ETuple();
// to
ETuple tmp = new ETuple(EVar var1, EVar var2, EVar var3,
                        EVar var4, EVar var5, EVar var6);
```

To create a new ETuple from an array of EVar:

```
ETuple tmp = new ETuple(EVar vars[]);
```

To retrieve the array of EVars:

```
EVar vars[] = tmp.value();
```

To retrieve the length of the Tuple:

```
int length = tmp.length();
```

To retrieve an enumeration of the EVars in the Tuple:

```
EVar evar[] = tmp.elements();
```

To retrieve an EVar at a specified index:

```
EVar evar = tmp.elementAt(1);
```

EProcess

The EProcess class is the Java wrapper for an Erlang Pid. The wrapper does not contain the Erlang Pid itself. Instead, it contains an internal integer value which allows the Erlang Jive server to find the corresponding Erlang Pid.

An EProcess class can be returned from the Erlang server, or as a result of spawning a process with the ESock module. An Erlang process can also be spawned automatically when creating an EProcess object.

```
EProcess p = new EProcess(String module, String name, EList Args);
EProcess p = new EProcess(EAtom module, EAtom name, EList Args);
```

> Warning:
> The creation of an EProcess object can throw a JiveException which *must* be caught.

EApplet

The EApplet class is a subclass of the Applet class. It will initiate the connection with the server and provide some useful methods, which are described below.

The EApplet class will connect to a port given by the port parameter, which must be specified within the <APPLET>, </APPLET> tags on the HTML page.

```
<APPLET CODE="...">
<PARAM NAME=PORT VALUE=4711>
</APPLET>
```

When creating a subclass based on EApplet, the `init()` method must be called in the EApplet class. The `init` method performs the connection with the server and creates the self Pid.

```
class Test extends EApplet {
  public void init() {
    super.init();
    /* Your code here! */
  }
}
```

By default, the Applet connects to the server when `start()` is called by the Applet, and disconnects from the server when `stop()` is called. To override these methods, you should call `super.start()` and `super.stop()` if you want the Applet default behaviour in addition to the customized functionality.

```
public void start() {
  super.start();
  /* Your code here! */
}
public void stop() {
  super.stop();
  /* Your code here! */
}
```

If you want to change the way the client connects/disconnects, do not call the `start()` and `stop()` methods in EApplet. Call the `connect()` and `disconnect()` methods instead.

The EApplet also provides six important methods:

`ESock getESock();` This is a method which returns the socket module (see The Run-time Environment [page 11]).

`EProcess self();` This is a method which returns the self Pid. This is an EProcess object used to send messages to Java from Erlang. It should be supplied to those Erlang processes/functions which send messages back to a Java client.

`EReceiver receiver();` This is a method which returns the EReceiver object. The EReceiver object is used to register a class implementation of the EReceive interface which makes it possible to receive messages sent from Erlang to a Java client.

`void connect();` This method connects to the server.

`void disconnect();` This method disconnects from the server.

`boolean connected();` This method checks if the Applet is connected to the server.

EApplication

The EApplication is used when building stand-alone Java Applications. The EApplication has the same functionality as EApple. Refer to the Jive examples [page 3] and the Java Jive API documentation for more details.

The Runtime Environment

The Jive runtime environment has the functionality to connect an Applet/Application to an Erlang server. The runtime environment is only instantiated once and can be retrieved with the static method `ERuntime runtime = ERuntime.getRuntime();`

At this time, the runtime environment only contains the socket module ESock, which is used for socket communication with the Erlang server. This will probably change in future versions of Jive.

The Socket Module

ESock is used for the socket communication with an Erlang server. The ESock object can be obtained by using the runtime environment, or by using a method in EApplet.

```
ERuntime runtime = ERuntime.getRuntime();
ESock eSock = runtime.getESock();
```

If inside an Applet:

```
ESock eSock = getESock();
```

Making an Erlang Apply

The ESock object provides methods for making an Erlang apply on the Erlang server side:

```
EVar reply = eSock.apply(String module, String name, EList args);
EVar reply = eSock.apply(EAtom module, EAtom name, EList args);
```

> Warning:
> An apply call can generate a JiveException which *must* be caught.

Spawning an Erlang Process

Erlang processes can be spawned by creating a new ESock module, or by creating a new EProcess object. The following two examples illustrate the different methods:

```
EProcess p = eSock.spawn(String module, String name, EList args);
EProcess p = eSock.spawn(EAtom module, EAtom name, EList args);

EProcess p = new EProcess(String module, String name, EList args);
EProcess p = new EProcess(EAtom module, EAtom name, EList args);
```

> Warning:
> Spawning of a process can generate a JiveException that *must* be caught.

Sending a Message to Erlang

Messages can be sent to Erlang processes by using functionality in the ESock class, or the EProcess classes. For obvious reasons, it is only possible to call processes which are represented by an EProcess object. The following examples illustrate:

```
eSock.send(EProcess p, EVar message);

p.send(EVar message);
```

> Warning:
> Sending a message can generate a JiveException which *must* be caught.

Receiving a Message from Erlang

Each class which needs to receive messages from the Erlang server must implement the REeceive interface. The method which has to be implemented is:

```
public void receive(EVar var);
```

In the following example, a class Test implements the EReceive interface and declares the receive method to print out the message if it is an Erlang string.

```
class Test implements EReceive {
  public void receive(EVar var) {
    if (var.type() == EVar.ESTRING) {
      String text = ((EString)var).value();
      System.out.println("Got string: "+text);
    }
  }
}
```

The EReceive object can be obtained with the `receiver()` method in EApplet:

```
EReceive test = (EReceive) new Test();
EReceiver receiver = receiver();
EInteger testID = receiver.register(test);
```

The EInteger, together with the self Pid, are supplied to the Erlang process to enable it to send messages that will be received by the Test object.

```
new EProcess ("module","function",new EList(testID,self()));
```

1.4 The Erlang Interface

This section describes the Erlang Interface to Jive and includes the following topics:

- Starting the Jive server
- Access control
- Using processes
- Using strings
- Sending a message to Java
- Detecting client disconnect.

Starting the Jive Server

Jive contains a module which handles the interaction between Java and Erlang. Before using any functionality in the `jive` module, a Jive server must be started. By default, this server listens to port 14000, but a different port can be supplied.

To start a server on port 14000, enter:

```
jive:start( ).
```

To start a server on port 4711, enter:

```
jive:start(4711).
```

Access Control

With the Java interface, it is possible to do Erlang `apply` on functions, or to spawn processes on the Erlang server. A strict access policy is used to prevent security problems. It is only possible to access functions which are explicitly declared from the Erlang side.

To allow Java to access the function `io:format/1`:

```
jive:allow({io,format,1})
```

To allow access to everything:

```
jive:allow(all)
```

> Warning:
> Use the `all` argument with care.

Using Processes

In order for Java to send messages to a specific Erlang process, the process must register its Pid and receive a corresponding Pid-Id.

```
PidId = jive:register_pid(Pid).
```

To get a Pid from a Pid-Id:

```
Pid = jive:get_pid(PidId).
```

Using Strings

Erlang does not differentiate between lists and strings. In Java, this differentiation is necessary for reasons of efficiency and clarity, and the `jive` module contains converters which convert between strings and lists.

A jive string is represented by the tuple {`string, [<char>]`}.

Conversion functions supplied are:

```
String = jive:list_to_string(List).
List = jive:string_to_list(String).
```

Sending a Message to Java

When a Java client connects to the Jive server, the server starts an Erlang process which allows the Jive server to send messages to it.

The Java client knows this Erlang process as the self process. The Java client must supply the self Pid-id to each process that should send messages back to the client.

The following example illustrates:

```
new EProcess("example","test",new EList(receiverID, self()));
```

A new Erlang process test is spawned with the self Pid-id as a parameter. The receiverID is an EInteger which specifies the Java object that should receive the message (see The Run-time Environment [page 11]).

The test process can then use the `PidId` to send messages to the client:

```
test(Receiver, PidId) ->
  Pid = jive:get_pid(PidId),
  Pid ! {send, Receiver, "This is a reply"}.
```

> Note:
> Messages sent to the client are wrapped into the tuple {`send, <receiver>, <message>`}.

Detecting Client Disconnect

The self process automatically terminates when a client disconnects. Before termination it removes itself from the jive process registry.

If you need to detect when the client disconnects, then link your process to the self process.

1.5 Limitations

This section describes the following limitations in Jive:

- Multiple back channels
- Proper non-blocking IO.

Multiple Back Channels

Java Applets are not allowed to create listen sockets which can accept socket connections. To enable messages to be sent from Erlang to Java, a socket connection is therefore kept open as long as the client is running.

When the client connects to the server, it opens a socket connection from the standard server port (14000). The client then sends a message through this port that causes the server to spawn a client proxy process, with the socket as a parameter. This proxy process is what is called the self-process. It accepts messages and sends those messages through the socket to the client.

This works well if there is little message traffic from the server to the client. However, all messages from the server are sent through this single socket and performance will suffer if the amount of data increases.

To solve this problem, the client can be allowed to open more than one socket connection to the server. This will enable the self process to send messages through different sockets simultaneously, which will increase message throughput.

Proper Non-Blocking IO

When waiting for input from a socket, the thread should be suspended until some input arrives according to the Java specification. Accordingly, waiting for input only suspends a single thread, not the entire runtime system.

This works fine with Solaris 1.x/2.x, but there are bugs in several implementations, including Netscape for Windows 95 and Netscape for Apple Macintosh. These implementations suspend the entire Java runtime system instead.

Temporary versions of non-blocking socket reads have been implemented as a work-around to this problem. The thread is suspended for a fixed amount of time if there is not enough input available, and checks are performed until there is enough input from the socket.

```
while (input.available() == 0) {
  try {
    Thread.sleep(100);
  } catch (InterruptedException e) {}
}
return input.read();
```

The delay is set to 100 ms, but it could be shorter. If the delay is too long it will slow down the message passing notably. Setting the value too short will cost too much processing power.

This should be changed as soon as these bugs are fixed.

1.6 JIVE Release Notes

This document describes the changes made to the Jive Application.

JIVE 1.3.1

Fixed Bugs and Malfunctions

- Reimplemented some undocumented application options to make version 1.3 more compatible with version 1.2.
- The start(Port) function ignored the port given and used the default port anyway. This is now corrected.

JIVE 1.3

Fixed Bugs and Malfunctions

- Jive startad for each connection a new process in a way that leaked memory. This is now corrected.
  Own Id: OTP-2829
- Jive now handles system messages.
  Own Id: OTP-2921
- There was a problem with Jive terminating when the user logs out in Windows NT. This was a problem with the old socket application and because Jive doesn't use it any longer the problem is gone.
  Own Id: OTP-3079
- A bug in the Netscape Java-VM that caused an exception in Jive. There is now a work around in Jive for this problem.
  Own Id: OTP-3110
- Jive did incorrectly handle the TCP stream incorrectly and this could cause random crashes, especially when there was high load on the machine. This is now corrected.
  Own Id: OTP-3112

Incompatibilities

- When the Erlang side starts a process as requested by the Java side it now link the Erlang communication process to the created process. This is to avoid the possibility of zombies in the system, i.e. processes that has no supervision.

JIVE 1.2

Fixed Bugs and Malfunctions

- Jive used the obsolete 'sockets' application for socket communication. It now uses 'gen_tcp'.
  Own Id: OTP-2858
- Added the 'reuseaddr' option to listen().
  Own Id: OTP-2904

JIVE 1.1

Fixed Bugs and Malfunctions

- Messages sent from Erlang to Java are now guaranteed to arrive in the same order as they were sent.
  Own Id: OTP-2093,seq818

JIVE 1.0.1.1

Fixed Bugs and Malfunctions

- Jive causes CRASH REPORT when netscape is disconnected
  Own Id: OTP-2038,seq538
- Jive now has a correct CXC number.
  Own Id: OTP-2187,seq657

Jive 1.0.1

Improvements and New Features

- The function `unsignedValue` has been added the the `EInteger` class. It returns an unsigned long version of the integer.
  `Own Id: OTP-1553`

Fixed Bugs and Malfunctions

- Zoombie Erlang processes was left when an Applet was terminated.
  `Own Id: OTP-1692`

Jive 1.0

Jive is a new Application allowing Java Applets/Applications to interact with Erlang. See the *Jive User's Guide*.

Java Interface Application (JIVE)

Short Summaries

- Erlang Module jive [page 22] – Java-Erlang Interface

jive

The following functions are exported:

- `start()`
  [page 22] Starts the Jive server at a specific port (default: 14000).
- `start(Port) -> pid()`
  [page 22] Starts the Jive server at a specific port (default: 14000).
- `stop() -> shutdown`
  [page 22] Stop the Jive server.
- `allow(all)`
  [page 22] Allows specific functions to be accessed by Java clients.
- `allow({Module,Function,Arity}) -> ServerRet`
  [page 22] Allows specific functions to be accessed by Java clients.
- `register_pid(Pid) -> ServerRet`
  [page 22] enables a Java client to send messages to a specific process.
- `unregister_pid(Pid) -> {remove,Pid}`
  [page 23] inhibits Java from sending messages to a specific process.
- `get_pid(PidId) -> ServerRet`
  [page 23] Converts a Pid to a PidId.
- `list_to_string(List) -> {string,String}`
  [page 23] Converts a list to a string.
- `string_to_list({string,String}) -> List`
  [page 23] Converts a string to a list.

jive (Module)

Jive is the glue which allows Java Applets/Applications to interact with Erlang.

Java and Erlang have Jive packages which hide the socket communication from the programmer. The Java side also contains a number of wrapper classes for each Erlang variable type (see the Java Jive API).

Java Applets/Applications can interact with Erlang and:

- spawn new Erlang processes
- send messages to Erlang processes
- do Erlang apply on functions.

Erlang processes can also send messages to Java objects.

Exports

`start()`
`start(Port) -> pid()`

Types:

- Port = int()

`start` starts the Jive server, listening to `Port`. The default port for `start/0` is 14000.

`stop() -> shutdown`

`stop/0` stops the Jive server.

`allow(all)`
`allow({Module,Function,Arity}) -> ServerRet`

Types:

- Module = Function = atom()
- Arity = int()
- ServerRet = all | {allow,{Module,Function,Arity}}

`allow/1` specifies from which functions processes can be spawn, or which functions to use in apply statements.

`register_pid(Pid) -> ServerRet`

Types:

- Pid = pid()
- ServerRet = error | {pidid,int()}

register_pid/1 enables Java to send messages to a specific Erlang process. The pidid is the Java client handle to an Erlang process.

unregister_pid(Pid) -> {remove,Pid}

Types:

- Pid = pid()

register_pid/1 inhibits Java from send messages to a specific Erlang process.

get_pid(PidId) -> ServerRet

Types:

- PidId = int()
- Pid = pid()
- ServerRet = error | Pid

get_pid/1 gets the actual Pid associated with a PidId. The PidId is the Java client handle to an Erlang process.

list_to_string(List) -> {string,String}

Types:

- List = String = string()

list_to_string/1 converts a list to a string. Java differentiates between lists and strings to achieve efficiency and clarity.

string_to_list({string,String}) -> List

Types:

- String = List = string()

string_to_list/1 converts a string to a list. Java differentiates between lists and strings to achieve efficiency and clarity.

Configuration

It is possible to start a Jive server using the following directives in a jive config file. This is used on embedded systems.

```
{jive,[{args,[Port,AllowedFunctions]}]}.
```

```
Port = int()
AllowedFunctions = [{Module,Function,Arity}]
Module = Function = atom()
Arity = int()
```

The Jive server listens to `Port` and `AllowedFunctions` specifies from which functions processes can be spawn, or which functions to use in apply statements (see allow/0 [page 22] above).

The following example shows a jive.config file:

```
[{jive,[{args,[14000,[{clock,start,2},{io,format,1}]]}]}].
```

The Jive server is started and listens to port 14000, allowing `clock:start/2` and `io:format/1` to be used by Java clients.

SEE ALSO

ig

List of Figures

Index

Modules are typed in *this way*.
Functions are typed in `this way`.