

jinterface

Copyright © 2000-2017 Ericsson AB. All Rights Reserved.
jinterface 1.6.1
December 1, 2017

Copyright © 2000-2017 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

December 1, 2017

1 Jinterface User Guide

The Jinterface Application is a java communication tool package to erlang.

1.1 The Jinterface Package

The *Jinterface* package provides a set of tools for communication with Erlang processes. It can also be used for communication with other Java processes using the same package, as well as C processes using the Erl_Interface library.

The set of classes in the package can be divided into two categories: those that provide the actual communication, and those that provide a Java representation of the Erlang data types. The latter are all subclasses of OtpErlangObject, and they are identified by the OtpErlang prefix.

Since this package provides a mechanism for communicating with Erlang, message recipients can be Erlang processes or instances of com.ericsson.otp.erlang.OtpMbox, both of which are identified with pids and possibly registered names. When pids or mailboxes are mentioned as message senders or recipients in this section, it should assumed that even Erlang processes are included, unless specified otherwise. The classes in *Jinterface* support the following:

- manipulation of data represented as Erlang data types
- conversion of data between Java and Erlang formats
- encoding and decoding of Erlang data types for transmission or storage
- communication between Java nodes and Erlang processes

In the following sections, these topics are described:

- mapping of Erlang types to Java
- encoding, decoding, and sending Erlang terms
- connecting to a distributed Erlang node
- using nodes, mailboxes and EPMD
- sending and receiving Erlang messages and data
- remote procedure calls
- linking to remote processes
- compiling your code for use with Jinterface
- tracing message flow

1.1.1 Mapping of Basic Erlang Types to Java

This section describes the mapping of Erlang basic types to Java.

Erlang type	Java type
atom	*OtpErlangAtom*
binary	*OtpErlangBinary*
floating point types	*OtpErlangFloat*or *OtpErlangDouble*, depending on the floating point value size

integral types	One of *OtpErlangByte*,*OtpErlangChar*,*OtpErlangShort*,*OtpErlangUShort*,*OtpI* depending on the integral value size and sign
list	*OtpErlangList*
pid	*OtpErlangPid*
port	*OtpErlangPort*
ref	*OtpErlangRef*
tuple	*OtpErlangTuple*
map	*OtpErlangMap*
term	*OtpErlangObject*

Table 1.1: Mapping of Erlang basic types to Java

1.1.2 Special Mapping Issues

The atoms `true` and `false` are special atoms, used as boolean values. The class *OtpErlangBoolean* can be used to represent these.

Lists in Erlang are also used to describe sequences of printable characters (strings). A convenience class *OtpErlangString* is provided to represent Erlang strings.

1.1.3 Nodes

A node as defined by Erlang/OTP is an instance of the Erlang Runtime System, a virtual machine roughly equivalent to a JVM. Each node has a unique name in the form of an identifier composed partly of the hostname on which the node is running, e.g "gurka@sallad.com". Several such nodes can run on the same host as long as their names are unique. The class *OtpNode* represents an Erlang node. It is created with a name and optionally a port number on which it listens for incoming connections. Before creating an instance of *OtpNode*, ensure that Epmd is running on the host machine. See the Erlang documentation for more information about Epmd. In this example, the host name is appended automatically to the identifier, and the port number is chosen by the underlying system:

```
OtpNode node = new OtpNode("gurka");
```

1.1.4 Mailboxes

Erlang processes running on an Erlang node are identified by process identifiers (pids) and, optionally, by registered names unique within the node. Each Erlang process has an implicit mailbox that is used to receive messages; the mailbox is identified with the pid of the process.

Jinterface provides a similar mechanism with the class *OtpMbox*, a mailbox that can be used to send and receive messages asynchronously. Each OtpMbox is identified with a unique pid and , optionally, a registered name unique within the *OtpNode*.

Applications are free to create mailboxes as necessary. This is done as follows:

```
OtpMbox mbox = node.createMbox();
```

The mailbox created in the above example has no registered name, although it does have a pid. The pid can be obtained from the mailbox and included in messages sent from the mailbox, so that remote processes are able to respond.

An application can register a name for a mailbox, either when the mailbox is initially created:

```
OtpMbox mbox = node.createMbox("server");
```

or later on, as necessary:

```
OtpMbox mbox = node.createMbox();
mbox.registerName("server");
```

Registered names are usually necessary in order to start communication, since it is impossible to know in advance the pid of a remote process. If a well-known name for one of the processes is chosen in advance and known by all communicating parties within an application, each mailbox can send an initial message to the named mailbox, which then can identify the sender pid.

1.1.5 Connections

It is not necessary to explicitly set up communication with a remote node. Simply sending a message to a mailbox on that node will cause the OtpNode to create a connection if one does not already exist. Once the connection is established, subsequent messages to the same node will reuse the same connection.

It is possible to check for the existence of a remote node before attempting to communicate with it. Here we send a ping message to the remote node to see if it is alive and accepting connections:

```
if (node.ping("remote",2000)) {
  System.out.println("remote is up");
}
else {
  System.out.println("remote is not up");
}
```

If the call to ping() succeeds, a connection to the remote node has been established. Note that it is not necessary to ping remote nodes before communicating with them, but by using ping you can determine if the remote exists before attempting to communicate with it.

Connections are only permitted by nodes using the same security cookie. The cookie is a short string provided either as an argument when creating OtpNode objects, or found in the user's home directory in the file `.erlang.cookie`. When a connection attempt is made, the string is used as part of the authentication process. If you are having trouble getting communication to work, use the trace facility (described later in this document) to show the connection establishment. A likely problem is that the cookies are different.

Connections are never broken explicitly. If a node fails or is closed, a connection may be broken however.

1.1.6 Transport Factory

All necessary connections are made using methods of *OtpTransportFactory* interface. Default OtpTransportFactory implementation is based on standard Socket class. User may provide custom transport factory as needed. See java doc for details.

1.1.7 Sending and Receiving Messages

Messages sent with this package must be instances of *OtpErlangObject* or one of its subclasses. Message can be sent to processes or pids, either by specifying the pid of the remote, or its registered name and node.

In this example, we create a message containing our own pid so the echo process can reply:

```
      OtpErlangObject[] msg = new OtpErlangObject[2];
      msg[0] = mbox.self();
      msg[1] = new OtpErlangAtom("hello, world");
      OtpErlangTuple tuple = new OtpErlangTuple(msg);
```

When we send the message, a connection will be created:

```
      mbox.send("echo", "gurka@sallad.com", tuple);
```

And here we receive the reply:

```
      OtpErlangObject reply = mbox.receive();
```

Messages are sent asynchronously, so the call to `send()` returns as soon as the message has been dispatched to the underlying communication layer. This means that you receive no indication whether the operation completed successfully or the remote even existed. If you need this kind of confirmation, you should wait for a response from the remote process.

The echo server itself might look like this:

```
OtpNode self = new OtpNode("gurka");
OtpMbox mbox = self.createMbox("echo");
OtpErlangObject o;
OtpErlangTuple msg;
OtpErlangPid from;

while (true) {
  try {
    o = mbox.receive();
    if (o instanceof OtpErlangTuple) {
      msg = (OtpErlangTuple)o;
      from = (OtpErlangPid)(msg.elementAt(0));
      mbox.send(from,msg.elementAt(1));
  }
  catch (Exception e) {
    System.out.println("" + e);
  }
}
```

In the examples above, only one mailbox was created on each node. however you are free to create as many mailboxes on each node as you like. You are also free to create as many nodes as you like on each JVM, however because each node uses some limited system resources such as file descriptors, it is recommended that you create only a small number of nodes (such as one) on each JVM.

1.1.8 Sending Arbitrary Data

This package was originally intended to be used for communicating between Java and Erlang, and for that reason the send and receive methods all use Java representations of Erlang data types.

However it is possible to use the package to communicate with remote processes written in Java as well, and in these cases it may be desirable to send other data types.

The simplest way to do this is to encapsulate arbitrary data in messages of type *OtpErlangBinary*. The OtpErlangBinary class can be created from arbitrary Java objects that implement the Serializable or Externalizable interface:

```
o = new MyClass(foo);
mbox.send(remote,new OtpErlangBinary(o));
```

The example above will cause the object to be serialized and encapsulated in an OtpErlangBinary before being sent. The recipient will receive an OtpErlangBinary but can extract the original object from it:

```
msg = mbox.receive();
if (msg instanceof OtpErlangBinary) {
   OtpErlangBinary b = (OtpErlangBinary)msg;
   MyClass o = (MyClass)(b.getObject());
}
```

1.1.9 Linking to Remote Processes

Erlang defines a concept known as linked processes. A link is an implicit connection between two processes that causes an exception to be raised in one of the processes if the other process terminates for any reason. Links are bidirectional: it does not matter which of the two processes created the link or which of the linked processes eventually terminates; an exception will be raised in the remaining process. Links are also idempotent: at most one link can exist between two given processes, only one operation is necessary to remove the link.

Jinterface provides a similar mechanism. Also here, no distinction is made between mailboxes and Erlang processes. A link can be created to a remote mailbox or process when its pid is known:

```
mbox.link(remote);
```

The link can be removed by either of the processes in a similar manner:

```
mbox.unlink(remote);
```

If the remote process terminates while the link is still in place, an exception will be raised on a subsequent call to receive():

```
try {
```

```
        msg = mbox.receive();
      }
      catch (OtpErlangExit e) {
        System.out.println("Remote pid " + e.pid() + " has terminated");
      }
      catch (OtpErlangDecodeException f) {
        System.out.println("Received message could not be decoded: " + f);
      }
```

When a mailbox is explicitly closed, exit messages will be sent in order to break any outstanding links. If a mailbox is never closed but instead goes out of scope, the objects `finalize()` method will call `close()`. However since Java provides no guarantees about when or even if finalize() will be called, it is important that your application explicitly closes mailboxes when they are no longer needed if you want links to work in a timely manner.

1.1.10 Using EPMD

Epmd is the Erlang Port Mapper Daemon. Distributed Erlang nodes register with epmd on the localhost to indicate to other nodes that they exist and can accept connections. Epmd maintains a register of node and port number information, and when a node wishes to connect to another node, it first contacts epmd in order to find out the correct port number to connect to.

The basic interaction with EPMD is done through instances of *OtpEpmd* class. Nodes wishing to contact other nodes must first request information from Epmd before a connection can be set up, however this is done automatically by *OtpSelf.connect()* when necessary.

When you use *OtpSelf.connect()* to connect to an Erlang node, a connection is first made to epmd and, if the node is known, a connection is then made to the Erlang node.

Java nodes can also register themselves with epmd if they want other nodes in the system to be able to find and connect to them. This is done by call to method *OtpEpmd.publishPort()*.

Be aware that on some systems (such as VxWorks), a failed node will not be detected by this mechanism since the operating system does not automatically close descriptors that were left open when the node failed. If a node has failed in this way, epmd will prevent you from registering a new node with the old name, since it thinks that the old name is still in use. In this case, you must unregister the name explicitly, by using *OtpEpmd.unPublishPort()*

This will cause epmd to close the connection from the far end. Note that if the name was in fact still in use by a node, the results of this operation are unpredictable. Also, doing this does not cause the local end of the connection to close, so resources may be consumed.

1.1.11 Remote Procedure Calls

An Erlang node acting as a client to another Erlang node typically sends a request and waits for a reply. Such a request is included in a function call at a remote node and is called a remote procedure call. Remote procedure calls are supported through the class *OtpConnection*. The following example shows how the *OtpConnection* class is used for remote procedure calls:

```
OtpSelf self = new OtpSelf("client", "hejsan" );
OtpPeer other  = new OtpPeer("server@balin");
OtpConnection connection = self.connect(other);

connection.sendRPC("erlang","date",new OtpErlangList());
OtpErlangObject received = connection.receiveRPC();
```

`erlang:date/0` is just called to get the date tuple from a remote host.

1.1.12 Compiling and Loading Your Code

In order to use any of the *Jinterface* classes, include the following line in your code:

```
import com.ericsson.otp.erlang.*;
```

Determine where the top directory of your OTP installation is. You can find this out by starting Erlang and entering the following command at the Eshell prompt:

```
Eshell V4.9.1.2  (abort with ^G)
1> code:root_dir().
/usr/local/otp
```

To compile your code, make sure that your Java compiler knows where to find the file `OtpErlang.jar` which contains the package. This is done by specifying an appropriate `-classpath` argument on the command line, or by adding it to the `CLASSPATH` definition in your `Makefile`. The correct value for this path is `$OTPROOT/lib/jinterface`*Vsn*`/priv/OtpErlang.jar`, where `$OTPROOT` is the path reported by `code:root_dir/0` in the above example and *Vsn* is the version of Jinterface, for example `jinterface-1.2`

```
$ javac -classpath ".:/usr/local/otp/lib/jinterface-1.2/priv/OtpErlang.jar"
                   myclass.java
```

When running your program, you will also need to specify the path to `OtpErlang.jar` in a similar way.

```
$ java ".:/usr/local/otp/lib/jinterface-1.2/priv/OtpErlang.jar" myclass
```

1.1.13 Tracing

Communication between nodes can be traced by setting a system property before the communication classes in this package are initialized. The value system property "OtpConnection.trace" is the default trace level for all connections. Normally the default trace level is zero, i.e. no tracing is performed. By setting *OtpConnection.trace* to some non-zero value, the communication protocol can be shown in more or less detail. The valid values are:

- 0: no tracing is performed
- 1: only ordinary send and reg-send messages are shown
- 2: control messages such as link, unlink and exit are shown
- 3: connection setup (handshake) is shown
- 4: epmd requests are shown

Each level also includes the information shown by all lower levels.

2 Reference Manual

Jinterface is a Java communication tool package to Erlang.

jinterface

Erlang module

Jinterface Java package contains java classes, which help you integrate programs written in Java with Erlang. The reference documentation is available only on the web in Javadoc format.

> ### Note:
>
> By default, the `jinterface` library is only guaranteed to be compatible with other Erlang/OTP components from the same release as the `jinterface` library itself. See the documentation of the `OtpCompatRel` property in the Java API documentation.

See Also

Jinterface User's Guide