Ibis Portability Layer (IPL) Programmer's Manual

The Ibis Group

November 7, 2007

1 Introduction

NOTE: THIS DOCUMENT IS COMPLETELY OUT OF DATE AND MUST BE REWRITTEN.

Ibis is an efficient and flexible Java-based programming environment for Grid computing, in particular for distributed supercomputing applications. This manual describes the Ibis Portability Layer (IPL). It also describes how to run an Ibis application. It is available on-line at http://www.cs.vu.nl/ibis/progman. Ibis is described in several publications (see Section 5). Rather than giving a detailed overview of what each class and method does, the aim of this document is to describe how to actually use these classes and methods. The *javadoc* subdirectory of the Ibis installation provides documentation for each class and method (point your favorite browser to javadoc/index.html file of the Ibis installation). The Ibis API is also available on-line at http://www.cs.vu.nl/ibis/javadoc. In this manual, fragments of an actual Ibis application will be used for illustration purposes. Section 3 will discuss a typical Ibis application, with subsections on each phase of the program. Section 4 will discuss how to actually compile and run this program.

2 Some Ibis concepts

2.1 The Ibis Portability Layer

The Ibis Portability Layer (IPL) consists of a set of Java interfaces and classes that define how an Ibis application can make use of the Ibis components. The Ibis application does not need to know which specific Ibis implementations are available. It just specifies some capabilities that it requires, and the Ibis system selects the best available Ibis implementation that meets these requirements.

2.2 An Ibis Instance

A loaded IPL implementation is called an *Ibis instantiation*, or *Ibis instance*. An Ibis instance is identified by a so-called *Ibis identifier*. An application can

find out which Ibis instances are present in the run by supplying a so-called *RegistryEventHandler*. This RegistryEventHandler is an object with, among others, a `joined()` method which gets called by the Ibis system when a new Ibis instance joins the run. The Ibis identifier of this new Ibis is a parameter to the `joined()` method.

2.3 Send Ports and Receive Ports

The IPL provides primitives to set up connections between send and receive ports. In general, a sendport can have connections to multiple receive ports, and a receive port can have multiple incoming connections. However, the user may specify that a sendport will have only one outgoing connection, or that a receive port will only have one incoming connection, allowing Ibis to choose a more efficient implementation. A connection is always *unidirectional*, from a send port to a receive port.

All send and receive ports have a *type* which is represented by an instance of `ibis.ipl.PortType`. To create a connection, an Ibis application must create a send port, and the instance at the receiving side must create a receive port. The send port can then be connected to the receive port, provided that the ports are of the same port type.

To send a message, the Ibis application requests a new write message from a send port. The write message object has methods to put data in this write message. Finally, the message is finished by invoking its `finish()` method.

To receive a message, the IPL provides two mechanisms:

explicit receipt when a receive port is configured for explicit receipt, a message can be received with the receive port's blocking *receive* method, or with its non-blocking *poll* method. These methods return a *read message* object, from which data can be extracted using its read methods. The *poll* method may also return *null*, in case no message is available.

upcalls when a receive port is configured for upcalls, the Ibis application provides an *upcall* method, which is to be called when a message arrives. The upcall provides the message received as a parameter. The message contents will be lost when the upcall returns, so the data in the message must be read in the upcall. Upcalls are either automatic, or must be polled for explicitly, see Section 3.1.1.

2.4 Port Types

Send and receive ports are *typed* by means of a *port type*. A port type is defined by capabilities. Port type capabilities that can be configured are, for instance, the serialization method used, reliability, whether a send port can connect to more than one receive port, whether more than one send port can connect to a single receive port, et cetera. Only ports of the same type can be connected.

2.5 Serialization

Serialization is a mechanism for converting Java objects into portable data that can be stored or transferred. Java has input (`java.io.ObjectInputStream`) and output (`java.io.ObjectOutputStream`) streams for reading and writing objects. Ibis has `writeObject` and `readObject` methods in its messages to accomplish the same effect.

Sometimes, object serialization is not needed. For that case, two simpler serialization mechanisms are available: *data serialization* which allows for sending/receiving data of basic types and arrays of basic types (similar to `java.io.DataInputStream` and `java.io.DataOutputStream`), and *byte serialization* which only allows sending/receiving bytes and arrays of bytes.

3 An Ibis Application

An Ibis application consists of several parts:

- Creating an Ibis instance in each instance of the application. An Ibis application can run on multiple hosts. On each of these hosts, an Ibis instance must be created.
- Setting up communication. Communication setup in Ibis consists of creating one or more *send ports*, through which messages can be sent, and creating one or more *receive ports*, through which messages can be received, and creating connections between them.
- Actually communicating. A send port is used to create a *write message*, which is sent to the receive ports that this send port is connected to.
- Finishing up. Connections must be closed, and each Ibis instance must be ended.

The next few subsections will discuss each of these steps in turn, illustrating them with parts of an RPC-style Ibis application. This application will have a server and one or more clients. This is a toy application: the server will receive strings, compute the length of each string, and send this length back to the client. The clients will send strings, and receive the length of these strings. The server will have to do some other work as well, just to make things a little more interesting.

3.1 Program Preamble

Ibis applications need to import classes from the IPL (Ibis Portability Layer) package, which lives in `ibis.ipl`. The easiest is probably to simply import all `ibis.ipl` classes with one import line:

```
import ibis.ipl.*;
```

Of course it is also possible to import only the needed classes.

Before creating an Ibis instance, we must determine what we want from it. This is determined by two objects: the required Ibis capabilities, and the list of port types that are going to be used.

3.1.1 Creating an Ibis Instance

All instances of a program that want to participate in an Ibis run must create an Ibis instance. To create an Ibis instance, the static `createIbis()` method of the `ibis.ipl.IbisFactory` class must be used. The specification of this method is:

```
static Ibis createIbis(IbisCapabilities requiredCapabilities,
                       RegistryEventHandler handler,
        PortType... portTypes)
         throws IbisCreationFailedException;
```

There may be several Ibis implementations available, and the factory selects the best one for you, based on the specified required capabilities and port types. These required capabilities and port types in fact determine what the user wants the chosen Ibis to be able to do. The Ibis factory will select an Ibis implementation that can at least do what is required, and if no such implementation is found, throw an exception. The possible Ibis capabilities are specified in `ibis.ipl.IbisCapabilities` class, as discussed in Section 3.1.2.

A registry event handler may be specified, with a.o. `joined()` and `left()` upcalls that get called when an Ibis instance joins or leaves the run. In our RPC example, we will not use this, so we specify `null` instead. However, when such a `RegistryEventHandler` is used, its `joined()` upcall is called for every Ibis that joins the run, including the Ibis being created itself. Upcalls to the `RegistryEventHandler` must be explicitly enabled by invoking the `enableRegistryEvents()` method of the Ibis just created. This ensures that the `RegistryEventHandler` has been able to do the necessary initializations.

The `enableResizeUpcalls()` method blocks until the `joined()` upcall for this Ibis has been invoked. Knowing which Ibises have joined the run, and how many there are, is often useful in dividing the work. See also section 3.2.1.

The last parameters specify the port types that are going to be used in the application. We will discuss these in more detail in Section **??**.

Now back to our example. In Secion 3.1.2 we will discuss the creation of an `IbisCapabilities` object describing the required capabilities. For the moment we will just assume it exists. Likewise, in Section **??** we will discuss the creation of a port type, and for now we will just assume that it exists, and is referred to by a variable `porttype`.

```
Ibis ibis = null;
try {
    ibis = Ibis.createIbis(capabilities, null, porttype);
} catch(IbisCreationFailedException e) {
    System.err.println("Could not create Ibis: " + e);
    ...
}
```

The method `createIbis()` throws an exception when no matching Ibis implementation can be found, or a matching Ibis cannot be instantiated for some reason.

3.1.2 Ibis Capabilities

Below is a snippet from the RPC example, constructing the required capabilities:

```
IbisCapabilities capabilities = new IbisCapabilities(
    IbisCapabilities.ELECTIONS_STRICT);
```

This states that the selected Ibis must support reliable elections (the toy example will use this to determine who will be the server and who will be the client).

This states that the selected Ibis must support reliable one-to-one communication, must support upcalls at the receiver side without the receiver having to poll for messages, and must also support explicit receipt of messages at the receiver side. (We want upcalls so that the server can do other work, and we want explicit receipt for the client side). In addition, the selected Ibis must support some form of object serialization (a string must be sent), and must support the "closed" worldmodel, which means that all participating Ibises join at the start of the run.

The complete list of predefined capabilities is given in the `PredefinedCapabilities` class. The most important ones are:

CONNECTION_ONE_TO_ONE One-to-one (unicast) communication is supported (if an Ibis implementation does not support this, you may wonder what it *does* support).

CONNECTION_ONE_TO_MANY one-to-many (multicast) communication is supported (in Ibis terms: a sendport may connect to multiple receiveports).

CONNECTION_MANY_TO_ONE many-to-one communication is supported (in Ibis terms: multiple sendports may connect to a single receiveport).

COMMUNICATION_FIFO messages from a send port are delivered to the receive ports it is connected to in the order in which they were sent.

CONNECTION_DOWNCALLS connection downcalls are supported. This means that the user can invoke methods to see which connections were lost or created.

CONNECTION_DOWNCALLS connection upcalls are supported. This means that an upcall handler can be installed that is invoked whenever a new connection arrives or a connection is lost.

COMMUNICATION_NUMBERED all messages originating from any send port of a specific port type have a sequence number. This allows the application to do its own sequencing.

COMMUNICATION_RELIABLE reliable communication is supported, that is, a reliable communication protocol is used. When not specified, an Ibis implementation may be chosen that does not explicitly support reliable communication.

RECEIVE_AUTO_UPCALLS upcalls are supported and polling for them is not required. This means that when the user creates a receiveport with an upcall handler installed, when a message arrives at that receive port, this upcall handler is invoked automatically.

RECEIVE_POLL_UPCALLS upcalls are supported but polling for them may be needed. When an Ibis implementation claims that it supports this, it may also do RECEIVE_AUTO_UPCALLS, but polling does no harm. When an application asks for this (and not RECEIVE_AUTO_UPCALLS), it must poll.

RECEIVE_EXPLICIT explicit receive is supported. This is the alternative to upcalls for receiving messages.

SERIALIZATION_BYTE Only the methods `readByte()`, `writeByte()`, `readArray(byte[])` and `writeArray(byte[])` are supported.

SERIALIZATION_DATA Only `read()`/`write()` and `readArray()`/`writeArray()` of primitive types are supported.

SERIALIZATION_OBJECT Some sort of object serialization is supported. This requires user-defined `writeObject()`/`readObject()` methods to be symmetrical, that is, each write in `writeObject()` must have a corresponding read in `readObject()` (and vice versa).

SERIALIZATION_STRICTOBJECT Sun serialization (through `java.io.ObjectOutputStream/InputS` is supported. In some regards, this object serialization implementation is more forgiving than SERIALIZATION_OBJECT.

WORLDMODEL_OPEN Ibises can join and leave the run at any time during the run.

WORLDMODEL_CLOSED The number of nodes involved in the run is known in advance and available from the `Ibis.totalNrOfIbisesInPool()` method.

If a specific implementation of Ibis is required, that can be dealt with too. There is a property called `ibis.name`, which can be used to supply the classname of the required Ibis implementation.

3.2 Setting up Communication

Setting up communication consists of several steps:

- create a port type;

- create a send and a receive port;
- set up connections between them.

The next few subsections discuss each of these steps in turn, but first we will discuss how to decide which Ibis instance does what.

3.2.1 Which Instance Does What?

Up until now, we have discussed only matters that all instances of the Ibis application should do, but now things become different. One instance of the application may want to send messages, while another instance may want to receive them. It may not even be clear which instance is going to do what. This can of course be solved with program parameters, but Ibis also provides a so-called registry (of type `ibis.ipl.Registry`), which is obtained through the `ibis.registry()` method. Ibis also provides the `ibis.ipl.IbisIdentifier` class. The `ibis.ipl.Ibis.identifier()` method returns such an Ibis identifier, which identifies this specific Ibis instance.

Using these methods it is possible to decide, in the RPC example, who is going to be the server by means of an "election": the Ibis registry provides a method `elect()` which (globally) selects one of a number of invokers. For our RPC example this could be done as follows:

```
IbisIdentifier me = ibis.identifier();
Registry registry = ibis.registry();
IbisIdentifier server = registry.elect("Server");
boolean iAmServer = server.equals(me);
IbisIdentifier client = null;
if (iAmServer) {
    client = registry.getElectionResult("Client");
} else {
    client = registry.elect("Client");
}
```

In our example, one instance of the program is the server and the other instance is a client. Of course, the client and the server can also be different programs altogether. Note that the server finds out who is the client by looking at an election result without being a candidate.

The `RegistryEventHandler`, as discussed in Section 3.1.1, can be used to keep track of the number of Ibis instances currently involved in the run.

3.2.2 Creating a Port Type

To be able to create send and receive ports, it is first necessary to create one or more *port types*. A port type is an object of type `ibis.ipl.PortType`. Within an Ibis instance, multiple port types, with different properties, can be created. The capabilities of a port type are, like the required capabilities of an Ibis implementation, specified by a `CapabilitySet` object. A port type is identified by these capabilities. The `Ibis` class contains a method to create a port type, specified as follows:

```
PortType createPortType(CapabilitySet portCapabilities);
```

For our RPC example program, we would create a port type with capabilities as discussed in Section 3.1.1:

```
CapabilitySet portCapabilities = new CapabilitySet(
    PredefinedCapabilities.CONNECTION_ONE_TO_ONE,
    PredefinedCapabilities.COMMUNICATION_RELIABLE,
    PredefinedCapabilities.SERIALIZATION_OBJECT,
    PredefinedCapabilities.RECEIVE_EXPLICIT,
    PredefinedCapabilities.RECEIVE_AUTO_UPCALLS);
PortType porttype = ibis.createPortType(portCapabilities);
```

In general, the port capabilities should be a subset of the capabilities specified when creating the Ibis instance. If a capability is specified that was not specified when creating the Ibis instance, this may result in an `IbisConfigurationException`.

3.2.3 Creating Send and Receive Ports

The `PortType` class contains several variants of a method `createSendPort()` that creates a send port (of type `ibis.ipl.SendPort`) and also several variants of a method `createReceivePort()` that creates a receive port (of type `ibis.ipl.ReceivePort`). See the API for an exhaustive list of variants.

In Ibis, receive ports usually have specific names, so that a send port can set up a connection to a receive port. In contrast, send ports usually are anonymous, because a receive port cannot initiate a connection.

For our RPC example, the server will have to create a receive port to receive a request and a send port to send an answer. The server is not allowed to block waiting for a request, so it will want a receive port that enables upcalls. To do that, the server must first define a class that implements the `ibis.ipl.MessageUpcall` interface. This interface contains one method:

```
void upcall(ReadMessage m) throws java.io.IOException;
```

We will go into the details of a `ReadMessage` in Section 3.4. For now, we will assume that there is a class `RpcUpcall` that implements this interface, and that the application has a field `rpcUpcall` of this type.

```
try {
    SendPort serverSender = porttype.createSendPort();
    ReceivePort serverReceiver =
        porttype.createReceivePort("server", rpcUpcall);
} catch(java.io.IOException e) {
    ....
}
```

The client will have to create a send port to send a request and a receive port to receive an answer. The client is allowed to block waiting for an answer, so it will want a receive port that enables explicit receipt. So, the client will create an anonymous server port, and a named receive port that enables explicit receipt (no upcall handler is supplied):

```
try {
    SendPort clientSender = porttype.createSendPort();
    ReceivePort clientReceiver =
        porttype.createReceivePort("client");
} catch(java.io.IOException e) {
    ....
}
```

When a receive port is created, it will not immediately accept connections. This must be explicitly enabled by invoking the `enableConnections()` method. Incoming connection attempts are kept pending until connections are enabled. So, the creator of the receive port can determine when he is ready to accept connections. If the receive port is configured for upcalls, these must explicitly be enabled by invoking the `enableMessageUpcalls()` method. Again, incoming messages are kept pending until upcalls are enabled.

3.2.4 Setting Up a Connection

Now that we have send ports and receive ports, it is time to set up connections between them. A connection is initiated by the `connect()` method of `ibis.ipl.SendPort`. Here is its specification:

```
void connect (IbisIdentifier ibis, String name) throws IOException;
```

This version blocks until an accept or deny is received. An `IOException` is thrown when the connection could not be established. There also is a `connect()` version with a time-out. So, we need a `IbisIdentifier` and a name to set up the connection. Here is the connection setup code for the server:

```
try {
    serverSender.connect(client, "client");
} catch(IOException e) {
    ...
}
```

Our RPC client would set up the following connection:

```
try {
    clientSender.connect(server, "server");
} catch(IOException e) {
    ...
}
```

This completes the connection setup.

Note that a send port can set up connections to more than one receive port (if the port type supports the `CONNECTION_ONE_TO_MANY` capability). Also, multiple send ports can set up connections to the same receive port (if the port type supports the `CONNECTION_MANY_TO_ONE` capability).

3.3 Connection upcalls, connection downcalls

Sometimes it is useful for an application to know which send ports are connected to a receive port, and vice versa, or which connections are being closed. Ibis implementations may support two different mechanisms for obtaining this type of information: connection upcalls and connection downcalls. See Section 3.1.2 for the corresponding capabilities.

When a port type is configured for using connection upcalls, a receive port may be instantiated with a `ReceivePortConnectUpcall` object. This is an interface with two methods:

```
boolean gotConnection(ReceivePort me,
                      SendPortIdentifier applicant);
void lostConnection(ReceivePort me,
                    SendPortIdentifier johndoe,
                    Exception reason);
```

The `gotConnection()` method gets called when a send port attempts to connect to the receive port at hand. An implementation of this method can decide whether to allow this connection or not by returning `true` or `false`. The `lostConnection()` method gets called when an existing connection to the receive port at hand gets lost for some reason.

A send port can be instantiated with a `SendPortDisconnectUpcall` object. This is an interface with a single method:

```
void lostConnection(SendPort me,
                    ReceivePortIdentifier johndoe,
                    Exception reason);
```

This method is called when an existing connection from the send port at hand gets lost for some reason. Note that there is no `gotConnection()` counterpart, because it is always the send port that initiates a connection.

When a port type is configured for using connection downcalls, receive ports and send ports of this type maintain connection information, and support methods that allow the user to obtain this information. A receive port has the following methods:

```
SendPortIdentifier[] newConnections();
SendPortIdentifier[] lostConnections();
SendPortIdentifier[] connectedTo();
```

`newConnections()` returns the send port identifiers of the connections that are new since the last call or the start of the program. `lostConnections()` returns the send port identifiers of the connections that were lost since the last call or the start of the program. `connectedTo()` returns the send port identifiers of all connections to this receive port. A send port has the following methods:

```
ReceivePortIdentifier[] newConnections();
ReceivePortIdentifier[] lostConnections();
ReceivePortIdentifier[] connectedTo();
```

which do exactly the same as their receive port counterparts.

3.4 Communicating

Communication in Ibis consists of messages, sent from a send port, and received at a receive port. When a sender wants to send a message, it will first have to obtain one from the send port. Such a message is of type `ibis.ipl.WriteMessage` and is obtained by means of the `newMessage()` method of `SendPort`, which is specified as follows:

```
WriteMessage newMessage() throws IOException;
```

For a given send port, only one message can be alive at any time. When a new message is requested while a message is alive, the request is blocked until the live message is finished.

Once a write message is obtained, data can be written to it. A write message has various methods for the different types of data that can be written to it. For instance, the `writeInt()` method can be used to write an integer value, and the `writeObject()` method can be used to write an object. The kind of data that can be written to the message depends on the serialization capability specified when the port type was created. The most general form is object serialization, which supports all write methods in a write message. The `data` serialization capability does not allow use of the `writeObject()` method, but does allow the use of all other write methods. The `byte` serialization capability only allows use of the `writeByte()` method and the `writeArray(byte[])` method.

Once there is a considerable amount of data in the message, Ibis can be given a hint to start sending, using the `send()` method. This hint is not required, however. When the message is complete, the message can be sent out by invoking the `finish()` method.

Our client in the RPC example could have the following:

```
int obtainLength(String s) throws IOException {
    WriteMessage w = clientSender.newMessage();
    w.writeObject(s);
    w.finish();
    ...
}
```

At the receiving side, a message can be received in two ways, depending on how the receive port was created: either by means of an upcall, or by means of an explicit receive. For each write method in the `WriteMessage` type, there is a corresponding read method in the `ReadMessage` type. For a given receive port, only one message can be alive at any time. A read message is alive until it is finished (by a `finish()` call), or the upcall returns.

Now, let us present some more code of our RPC example, this time from the server:

```
public void upcall(ReadMessage m) throws IOException {
    String s = "";
    try {
s = (String) m.readObject();
    } catch(ClassNotFoundException e) {
```

```
            ...
        }
        int len = s.length();
        m.finish();
        WriteMessage w = serverSender.newMessage();
        w.writeInt(len);
        w.finish();
    }
```

Note that the read message is finished before replying to the request. To prevent deadlocks, upcalls are not allowed to block (call `Thread.wait()`) or access the network (write a message or read another message) as long as a read message is active.

Now, we can also finish the `obtainLength()` method of the client:

```
        ...
        ReadMessage r = clientReceiver.receive();
        int len = r.readInt();
        r.finish();
        return len;
    }
```

3.5 Finishing up

Closing of a connection is initiated by closing a send port by means of the `close()` method. The `ReceivePort` class also has a `close()` method, but this method blocks until all send ports that have a connection to it are closed. So, send ports have to be closed first.

Our RPC client will do the following:

```
clientSender.close();
clientReceiver.close();
```

and the code of the server should be clear by now.

Ibis itself must also be ended. Both our client and our server should invoke the `Ibis.end()` method:

```
ibis.end();
```

As of Java 1.3, it is also possible to add a so-called shutdown hook. This could be done right after the Ibis instance is created:

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        try {
            ibis.end();
        } catch (IOException e) {
        }
    }
});
```

This shutdown hook gets invoked when the program terminates, and forcibly closes all ports.

3.6 Ibis utilities

The `ibis.util` package contains several utilities that may be useful for Ibis applications. The `ibis.util.Stats` utility contains methods for computing the mean and standard deviation of an array of numbers. A timer utility is provided in `ibis.util.Timer`. See the Ibis API for other utilities. Most of these are used in Ibis implementations, but may have other uses.

3.7 Avoiding deadlocks

As with most communication layers, it is quite easy to write code that deadlocks with Ibis. For example, if you have two Ibis instances that are writing large amounts of data to each other, and there are no readers for this data active, this will almost certainly result in a deadlock, because network buffers will fill up, causing the senders to block. Such a deadlock can be avoided by having a separate reader thread, or by installing an upcall handler for the incoming message.

Another common source of deadlocks is if you have a port type that specifies the `CONNECTION_MANY_TO_ONE` as well as the `CONNECTION_ONE_TO_MANY` capability. Multiple hosts doing simultaneous multicasts is a well-known source of deadlocks, because most systems do not implement a functioning flow-control for these cases.

4 Compiling and Running an Ibis Application

Before running an Ibis application it must be compiled. Using *ant*, this is quite easy. Assuming that the environment variable `IBIS_HOME` reflects the location of your Ibis installation, here is a `build.xml` file for our example program:

```
<project
    name="client-server"
    default="build"
    basedir=".">

    <description>
    Ibis application build.
    </description>

    <property environment="env" />
    <property name="ibis"   value="${env.IBIS_HOME}" />

    <property name="build"  location="build"/>

    <import file="${ibis}/build-files/apps/build-ibis-app.xml"/>
</project>
```

Now, invoking *ant* compiles the application, leaving the class files in a directory called `build`.

If, for some reason, it is not convenient to use *ant* to compile your application, or you have only class files or jar files available for parts of your application, it is also possible to first compile your application to class files or jar files, and then process those using the *ibisc* script. This script can be found in the Ibis bin directory. It takes either directories, class files, or jar files as parameter, and processes those, possibly rewriting them. In case of a directory, all class files and jar files in that directory or its subdirectories are processed.

4.1 The Ibis Registry

Most Ibis implementations depend on a registry server for providing information about a particular run, such as finding Ibis instances participating in the run, elections, et cetera. The Ibis registry server collects this information for multiple Ibis runs, even simultaneous ones. It does so by associating a user-supplied identifier with each Ibis run. Each Ibis instance announces its presence to the registry server, using this identifier, so that the registry server can determine to which Ibis run this Ibis instance belongs. The registry server then notifies the other Ibis instances of this run that a new instance has joined the run, including some identification of this instance.

If you tell Ibis that the registry server location is a machine that also participates in the run itself, Ibis will automatically try to start a registry server. How you can specify this is explained in the next section. If you want to run the registry server on a seperate host, one that is not behind a firewall, for instance, you have to start the registry server by hand.

The Ibis registry server is started with the `ibis-server` script which lives in the Ibis `bin` directory. Before starting an Ibis application, you need to have a registry server running on a machine that is accessible from all nodes participating in the Ibis run. The registry server expects the Ibis instances to connect to a socket that it creates when it starts up. The port number of this socket can be specified using the `--port` command line option to the `ibis-server` script.

4.2 Running an Ibis Application

An Ibis instance is started with the `ibis-run` script which lives in the Ibis `bin` directory. This `ibis-run` script is called as follows:

`ibis-run` *java-flags class params*

The `ibis-run` script is just a small script that adds the jar files from the Ibis lib directory to your classpath and then starts Java.

4.3 Running the example

You can run the example both with and without the `ibis-run` script. To run the application, we first need an ibis-server. Start a shell and run the `ibis-server` script:

```
$ $IBIS_HOME/bin/ibis-server
```

Then, you need to start two shells. In both shells type:

```
$ $IBIS_HOME/bin/ibis-run \
    -Dibis.server.address=localhost -Dibis.pool.name=bla \
    ibisApps.example.Example
```

Now, two instances of your application should run. One of them should print:

```
Test succeeded!
```

The `ibis.pool.name` value can be any random string. It identifies your run. This is done because a single Ibis registry server can serve multiple runs. The `ibis.server.address` property should be set to the machine you run the Ibis server on. In this case, we use `localhost`.

If you don't use the `ibis-run` script, you have to set the classpath and some additional properties. In both shells, type:

```
$ java \
    -cp $IBIS_HOME/lib/ipl.jar:$IBIS_HOME/lib/ipl-app.jar \
    -Dibis.impl.path=$IBIS_HOME/lib \
    -Dibis.server.address=localhost \
    -Dibis.pool.name=bla \
    -Dlog4j.configuration=file:$IBIS_HOME/log4j.properties \
    ibisApps.example.Example
```

The classpath specified here specifies the jar-files that are explicitly used: `ipl-app.jar` contains the example code, and `ipl.jar` contains the Ibis IPL. All other jar-files are loaded by the ibis classloader, which looks in the directory specified by the `ibis.impl.path` property, and its sub-directory, for any jar-file that could be needed.

5 Further Reading

The Ibis web page http://www.cs.vu.nl/ibis/publications.html contains links to various Ibis papers. The best starting point might be http://www.cs.vu.nl/ibis/papers/nieuwpoort_cpe_05.pdf, which gives a high-level description of the stucture of the Ibis system. It also gives some low-level and high-level benchmarks of the different Ibis implementations.

The *javadoc* subdirectory of the Ibis installation provides documentation for each class and method in the Ibis API (point your favorite HTML viewer to javadoc/index.html in the Ibis installation). The Ibis API is also available on-line at http://www.cs.vu.nl/ibis/api/index.html.