Ibis Communication Library Programmer's Manual

http://www.cs.vu.nl/ibis

November 12, 2009

1 Introduction

This manual explains how to program applications using the Ibis communication library. The Ibis communication library is a portable, high performance, Java based library. For information on how to run applications using this library, see the Ibis user's manual, available in the Ibis distribution in the `docs` directory, or on the Ibis website[1].

In this manual, we will focus on concepts and methods rather than details like exact parameters of methods. For this type of information, see the Javadoc for Ibis. This is available in the `javadoc` directory of the distribution, and on the website.

This manual is split up into several different sections. First, we will explain the different parts of the *Ibis Portability Layer* (IPL), the interface to the Ibis communication library. Second, we explain how to compile applications that use the IPL. Last, we will give some examples to show how to use Ibis in practice.

2 IPL Overview

This section will give a high level overview of the different parts of the IPL. For an overview of the design of Ibis, see Figure 1. The IPL offers applications a grid aware communication library, independent of the actual network used. Ibis dynamically selects a suitable implementation at run-time. The Ibis communication library ships with several different TCP based implementations, but implementations based on native platforms such as MPI and Myrinet are also possible.

Using the IPL, we implemented several different programming models, such as the divide-and-conquer Satin model, and the Group Method Invocation (GMI) model. A discussion of these models lies outside the scope of this manual. For more information on the programming models, see the Ibis website. As an alternative to using an intermediate programming model, applications can also be written directly on top of the IPL. This manual is mostly intended for programmers using the IPL in an application and for programming model writers.

A central concept in the IPL is the *Ibis instance*. In general, one Ibis instance is created on each machine participating in a particular distributed application. Each of

[1] `http://www.cs.vu.nl/ibis`

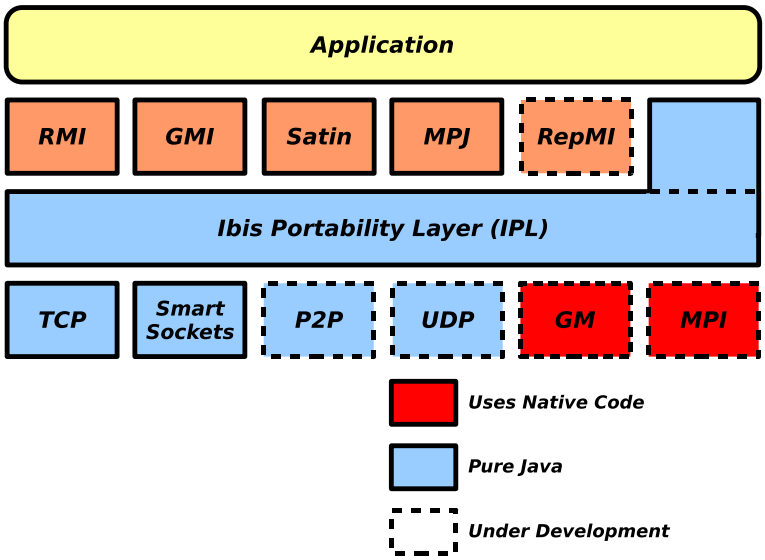

Figure 1: Ibis design overview

these instances has a unique *Ibis Identifier*, used throughout the library to denote this instance.

Another often-used concept is the *upcall*: a method that gets invoked upon an event such as the arrival of a message. Upcalls can be used instead of downcalls (explicit receipt, polling) in a number of cases. These cases include receiving messages, getting notifications of new connections, and updates on new Ibis instances joining a computation.

2.1 Send and Receive Ports

The IPL is based on uni-directional connection-oriented pipes. In the model, *send ports* are connected to *receive ports*. See Figure 2 for some examples. Connecting a single send port to a single receive port leads to a simple one-to-one connection. However, it is also possible to connect a send port to multiple receive ports for a one-to-many multicast, or multiple send ports to a single receive port for many-to-one client-server type communication. Even connecting multiple send ports to multiple receive ports is possible.

Ports in Ibis are typed (see Figure 3). Types are determined by the communication pattern (one-to-one, one-to-many, many-to-one, many-to-many), the type of data that can be sent, reliability, and other aspects. Send ports can only be connected to receive ports of the same type. This allows for some sanity checks on the application, and allows Ibis to select the most appropriate implementation of the communication

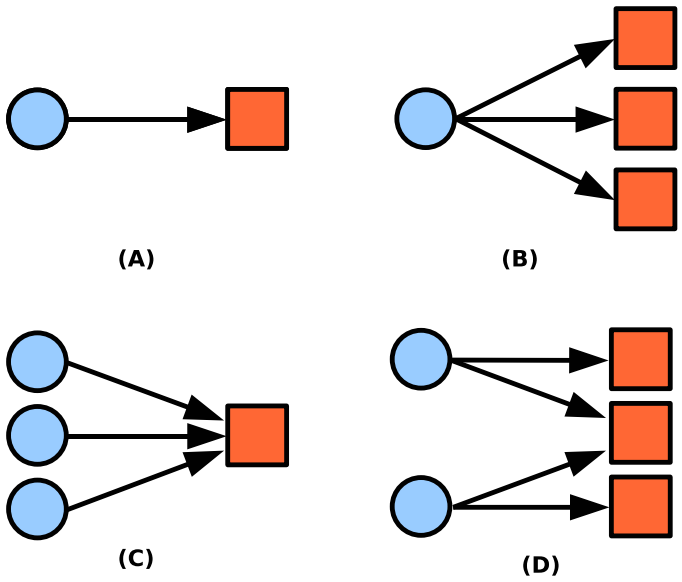

Figure 2: Ibis send and receive port configurations: one to one (a), one to many (b), many to one (c), and many to many (d)

channel.

The port type is determined by a set of capabilities as listed in Section 2.4.2. Note that even the configuration of the receive port as specified in the RECEIVE_XXX capabilities is part of the port type, and therefore must be specified in the capabilities of the send port.

2.2 Messages and Serialization

When a connection has been established between send and receive ports, data can be sent. Send and receive ports in Ibis communicate using messages. A so called *write message* is created at the send port, and can be filled with all primitive data types of Java such as bytes, integers and floats, as well as more complex data structures in the form of Java `Objects`. Also, arrays of primitive types and objects can be sent. When the user is done writing data to a message, a `finish` method must be called to denote that the message has been completed.

Once data is written to a write message at the send port, this message will arrive at the receive port as a *read message*. There, the data can be read again using methods equivalent to the ones in the write message. Data must be read from the read message in the same order as it was written in the write message.

In Ibis, the size of a message is unlimited. However, at any point in time, only a single message can be active at a port. This allows Ibis to start sending data to the

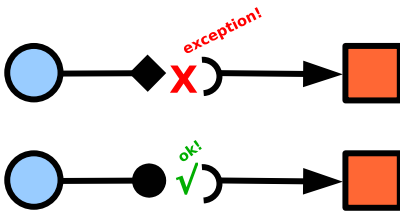

Figure 3: Types must match when connecting

receiver as soon as it is written in the message. So, while data is still being written to a message at the sender, it may already be read at the receiver. This streaming behavior makes communication in Ibis much more efficient.

Serialization is the translation of data objects into bytes suitable for sending across a network. Ibis is capable of several different versions of serialization, depending on the needs of the user. These versions differ in the types of data they support. Some only support sending bytes, some only primitive types, and some also allow sending objects. To be able send an Object, this object must implement the `java.io.Serializable` interface. See the documentation of Java for more information on serialization.

Two different object serialization versions are available in Ibis. One is the standard Java object serialization, the other is a high-performance implementation of object serialization. For this high-performance *Ibis Serialization* implementation to work, an extra step is needed when compiling the application, where some extra code is generated to efficiently send objects. See Section 3 for more information.

2.3 Pools and the Registry

Each Ibis instance is a member of a *Pool*. The Ibis *Registry* keeps track of all the members of the pool, and notifies all Ibises whenever a new Ibis instance joins this pool, leaves the pool, or even when an Ibis dies unexpectedly. Applications can use the registry mechanism to get notified of any changes to the pool.

The Ibis registry also supports *elections*, which can be used to select Ibis instances which are special, such as servers. Each election has a unique name in the form of a String, and will elect a winner from the candidates for this election. However, elections results are not determined by majority vote: usually the first candidate is elected as the winner. It is also possible to request the outcome of an election without being a candidate. When the winner of an election leaves the pool, or dies, a new winner is automatically selected if any new candidate enters the election.

The registry has a few other features useful for coordinating distributed computations. One is a *signal* system, where the registry can send simple strings to Ibises. This can be used to signal special events, such as the application terminating. Another feature is the *sequencer*. The sequencer can be used to get unique sequence numbers in a

distributed fashion. Each call to this method, at whatever Ibis instance, will yield the next number in the sequence. A central coordination point is used to implement this reliably.

The normal implementation of the Registry is a central registry service. This allows for a reliable system, with a good consistency model. Elections are guaranteed to yield the same winner at each Ibis, and any changes to the membership list of the pool are reported in the same order at each Ibis. In this model, joins and leaves are guaranteed to be totally ordered. This implementation does rely on a central server for coordination. This may lead to inefficiencies, and is also a single point of failure. So, as an alternative, other, less strict, consistency models are also offered. When these are used, Ibis is able to select an alternative implementation of the registry, for instance one based on Peer-to-Peer techniques.

2.4 Capabilities

To facilitate the selection of the best implementation for all the functionality required of Ibis from a user, Ibis uses a *capability* based selection mechanism. For each Ibis instance and each port type, the user can specify which capabilities are needed. Each capability enables a certain part of the API. For instance, sending objects in a message is only possible if the ”object serialization” capability is enabled in the port type. Likewise, elections can only be done if either the ”unreliable election” or the ”strict election” capability is enabled. The difference between the two being the consistency model of the elections.

Two different types of capabilities are present in Ibis. Some are global capapabilities of an Ibis instance, and some are capabilities of port types. Since different port types can be used in a single application, a user can specify different requirements for different ports.

2.4.1 Global Ibis Capabilities

This is a list of the capabilities for an Ibis. These capabilities are defined in the `IbisCapabilities` class of the IPL. Also see the Javadoc.

SIGNALS Enables the signal sending and receiving functionality of the registry.

MEMBERSHIP_UNRELIABLE Enables the membership information functionality of the registry. The registry can notify the user whenever a new Ibis joins the pool or a Ibis leaves the pool, or dies. This is not reliable though, and some changes to the pool may not be reported. Also, the order in which changes are reported is not fixed.

MEMBERSHIP_TOTALLY_ORDERED As above, but with a better consistency model. Each and every change to the pool will be reliably reported, and the order of these changes is fixed for the entire pool. All Ibises get the changes in the same, globally unique, order.

ELECTIONS_UNRELIABLE Enables unreliable elections. With unreliable elections it is possible that some election results are not available, and if the same

election is done at two different Ibises, the result may be different at one instance from the result at another instance.

ELECTIONS_STRICT Indicates that elections are supported, and they adhere to the following consistency model: Each election has a single, unique winner at any given point in time, and all Ibis instances in a pool get all updates to the election result, and in the same order.

CLOSED_WORLD The Ibises in the pool are determined at the start of the run. After these Ibises have joined, any further requests to join the pool will be denied. This capability also enables the usages of the `getPoolSize()` and `waitUntilPoolClosed()` methods in the `Registry` class.

This capability requires the number of Ibises in the pool to be known in advance. This can be set using a property (see Section 2.5).

MALLEABLE Requests malleability support from Ibis. Some implementations of Ibis (such as an Ibis based on MPI), may not support Ibises joining the pool after it has initially been created.

2.4.2 Port Type Capabilities

This is a list of capabilities for port types. These capabilities are defined in the `PortType` class of the IPL. For each port type, a single capability for the allowed connection types (one-to-one, multicast, etc) must be selected, as well as a type of serialization (bytes only, primitive types, objects, etc).

CONNECTION_ONE_TO_ONE One-to-one (unicast) communication is supported.

CONNECTION_ONE_TO_MANY One-to-many (multicast) communication is supported (in Ibis terms: a send port may connect to multiple receive ports).

CONNECTION_MANY_TO_ONE Many-to-one communication is supported (in Ibis terms: multiple send ports may connect to a single receive port).

CONNECTION_MANY_TO_MANY Many-to-one communication is supported (in Ibis terms: multiple send ports may connect to multiple receive ports).

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

CONNECTION_UPCALLS 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.

CONNECTION_TIMEOUT Enables the usage of timeouts in connection attempts.

RECEIVE_EXPLICIT Explicit receive is supported. A message can be received by directly requesting the reception of a message from the receive port.

RECEIVE_TIMEOUT Explicit receive with a timeout is supported.

RECEIVE_POLL Non-blocking version of explicit receipt is supported.

RECEIVE_AUTO_UPCALLS Upcalls are supported and polling for them is not required. This means that when the user creates a receive port 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 is required. So, a user must call the `poll` method in the receive port to reliably receive messages, but these messages will be delivered using an upcall.

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.

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.

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

SERIALIZATION_DATA Only `read()`, `write()`, `readArray()` and `writeArray()` of primitive types are supported in read an write messages.

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_OBJECT_SUN Sun serialization is supported through `java.io.ObjectOutputStream` and `java.io.ObjectInputStream`. Sun serialization is very good for debugging your application, as errors are more verbose than Ibis seerialization.

SERIALIZATION_OBJECT_IBIS Ibis serialization is supported. This is more efficient than sun serialization, but may have some limitations. For instance, versioning of objects is not supported.

2.5 Properties

Properties are used in Ibis for setting configuration options at run-time. They may be used to specify for example where to find the Ibis implementations, or the name of the pool this Ibis must join. Properties can be passed to an Ibis when it is created, but by default, properties specified on the command line are used. See the user's guide for more information on the various properties of Ibis and their usage.

3 Compiling applications

Applications that use Ibis can be compiled as any normal Java application. The file `ipl-2.2.jar` needs to be added to the classpath when compiling (and running) an Ibis application. For the serialization of Ibis to function correctly, one additional step is required after compiling: the code for writing and reading objects must be generated. The *Ibis compiler* application is available for this purpose. It can be run by using the `iplc` script in the `bin` directory of the distribution. The Ibis compiler needs to be passed the class files of the application, either as a directory containing classes, or as a jar file.

Ibisc can also be called from `ant`[2], the make-like build system for Java. An example of a build file which calls the Ibis compiler is available in the `examples` directory of the distribution. The relevant piece of the script is this:

```
<java classname="ibis.compile.Ibisc"
      taskname="Ibisc"
      failonerror="true"
      fork="true">
          <arg line="ipl-examples-2.2.jar" />
          <classpath refid="default.classpath" />
</java>
```

It runs the Ibis compiler on the jar file that has just been created from the compiled classes of the application. It sets the classpath to the `lib` directory of the Ibis distribution.

4 Tutorial

After giving a high level overview of the functionality of Ibis, we will now give some examples of applications which use the IPL. All the examples used here can also be found in the `examples` directory of the distribution.

4.1 Hello

The first example is a very simple Hello World type application. This is also the example used in the user's guide. So, try to get it to run to see what it is supposed to do. See Figure 4 for the complete source of the Hello application. It is under a hundred lines of code. This application is meant to be started twice. One instance will act as a server and one as a client. The client sends a message to the server. Which instance is the server and which is the client is determined using an election.

The application is split up into several parts. First, the capabilities needed from Ibis are defined in two global variables. At the very bottom of the file is the `main` method. Main creates a Hello object and invokes `run` on it. Run initializes Ibis, and

[2] `http://ant.apache.org`

```
1  package Ibis.ipl.examples;
2
3  import ibis.ipl.*;
4
5  public class Hello {
6      PortType portType = new PortType(PortType.COMMUNICATION_RELIABLE,
7              PortType.SERIALIZATION_DATA, PortType.RECEIVE_EXPLICIT,
8              PortType.CONNECTION_ONE_TO_ONE);
9
10     IbisCapabilities ibisCapabilities = new IbisCapabilities(
11             IbisCapabilities.ELECTIONS_STRICT);
12
13     private void server(Ibis myIbis) throws IOException {
14
15         // Create a receive port and enable connections.
16         ReceivePort receiver = myIbis.createReceivePort(portType, "server");
17         receiver.enableConnections();
18
19         // Read the message.
20         ReadMessage r = receiver.receive();
21         String s = r.readString();
22         r.finish();
23         System.out.println("Server received: " + s);
24
25         // Close receive port.
26         receiver.close();
27     }
28
29     private void client(Ibis myIbis, IbisIdentifier server) throws IOException {
30
31         // Create a send port for sending requests and connect.
32         SendPort sender = myIbis.createSendPort(portType);
33         sender.connect(server, "server");
34
35         // Send the message.
36         WriteMessage w = sender.newMessage();
37         w.writeString("Hi there");
38         w.finish();
39
40         // Close ports.
41         sender.close();
42     }
43
44     private void run() throws Exception {
45         // Create an ibis instance.
46         Ibis ibis = IbisFactory.createIbis(ibisCapabilities, null, portType);
47
48         // Elect a server
49         IbisIdentifier server = ibis.registry().elect("Server");
50
51         // If I am the server, run server, else run client.
52         if (server.equals(ibis.identifier())) {
53             server(ibis);
54         } else {
55             client(ibis, server);
56         }
57
58         // End ibis.
59         ibis.end();
60     }
61
62     public static void main(String args[]) {
63         try {
64             new Hello().run();
65         } catch (Exception e) {
66             e.printStackTrace(System.err);
67         }
68     }
69 }
```

Figure 4: Complete source of the Hello program

determines if this is the client or the server. It then calls either the `server` or the `client` method.

We will now explain this application line-by-line. The example starts with declaring the package on line 1. It then imports all classes from the IPL on line 3, and declares the `Hello` class on line 5. Since this application will send messages, it will need to create ports, and thus a port type. The declaration on lines 6-8 creates a port type suitable for this application.

The constructor of the `PortType` class requires us to specify which capabilities we want to set. We select `COMMUNICATION_RELIABLE`, which makes sure the message will arrive and `SERIALIZATION_DATA`, which allows us to send primitive types, not only bytes, since we want to send a string. `RECEIVE_EXPLICIT` denotes that we are going to explicitly call the `receive()` method of the receive port, and finally `CONNECTION_ONE_TO_ONE` selects the simplest communication pattern, where a single send port is connected to a single receive port. Next, we also need a list of all the capabilities we need from Ibis itself. Since we would like to use reliable elections, we include `ELECTIONS_STRICT`.

The main entry point of this class is the `main` method, defined on lines 62-67. It simply creates a object of the Hello type and calls the `run` method, defined on lines 44-60. The Ibis instance used by the applications is created using the `IbisFactory` class on line 46. Ibises cannot be created directly, but must be made using a call to this factory. The `createIbis` method takes several parameters. First is the `capabilities` object of this Ibis. Next, a `RegistryEventHandler` must be passed if needed. This example does not, see the Registry example below for this functionality. Lastly, the `createIbis` method requires a list of all the port types needed by the application. We only have one, which we defined previously.

When the Ibis instance is created, it is automatically added to the pool specified with the properties set at the command line (see the user's guide). So, we can now start using the registry, make connections, etc. This application starts by determining which instance will be the server. This is done via an election on line 49. Each election in Ibis has a name, denoted by a string. So, as each instance of this application calls the `elect` method of the registry with the same parameter ("Server" in this case), one of these Ibises will "win" this election. The election returns the winner. On line 52 we compare the winner with our own identifier. If we won the election, the `server` method is called on line 53, if we didn't win we are the client and we call the `client` method on line 55.

The `server` method, starting on line 13, waits for a message from the client and prints it. So, to receive a message we first have to create a receive port(line 16). Both send and receive ports are created by calling one of the `create` methods of the Ibis instance. In this case, we use the `createReceivePort` method which has two parameters: one for the port type, and one for the name of this receive port. This name must be unique for this Ibis instance, and can be used to connect to this port. We name our port "server".

After a receive port has been created, it does not automatically start handling new connections. This allows an application to initialize itself properly before any connections come in from other Ibises. Since we don't have anything else to initialize we simply enable incoming connections immediately using the `enableConnections`

method on line 17.

Next, we will receive the message. We specified we were going to use explicit receive in the capabilities of our port type, so we call the `receive` method on our receive port(line 20) to wait for a message. Once a message has been received we read the data from the message (we know it is a string) by calling the `readString` method on the read message. Afterwards, we signal Ibis we are now done with this message by calling `finish` on the message. Finally, we print the message on line 23. Since we are only receiving a single message our server is now done, and closes the receive port on line 26.

On the client side, we will have to connect to the server, and send it a message containing a string. So, we first create a send port on line 32 by calling one of the create methods of our Ibis instance. The method we call expects a port type (this must be equal to the port type of the receive port we want to connect to). After creating the port we connect our send port to the receive port. We use the identifier of the server we acquired at the election, and the name of the port at the server which we know to be "server".

After the port has been connected, we can now send a message. We ask the send port for a new write message on line 36, and write a string to it on line 37. Since we are only writing this one string we finalize the message on line 38. This completes the work of the client. So, we close our send port on line 41.

If everything worked as planned, the server should now print the message it received from the client. Both the server and the client exit their respective methods and return to the `run` method on line 57. On line 58 they both end their Ibis instance. This will make these two instances leave the pool, ending with a empty pool.

4.2 Upcalls

The next example we will look at is basically the same application as the first, except this application uses upcalls instead of explicit receive. Instead of listing the entire application, we only list the parts that need to be changed to use upcalls in Ibis in Figure 5. For the complete source, see the `examples` directory of the distribution.

On line 1, the first change is visible. The application extends the `MessageUpcall` interface from the IPL. This class contains a single method (`upcall`) which is called whenever a message is received by Ibis. If upcalls are used it is not necessary to call the `receive` method on a receive port. Instead, Ibis will wait for a message continually and pass any new messages received to the upcall handler provided by the user, one by one.

Next, the port type needs to be changed slightly. We no longer include the RECEIVE_EXPLICIT capability. Instead we use the RECEIVE_AUTO_UPCALLS capability. The "auto" in this capability means Ibis can receive messages automatically, without any user intervention. The RECEIVE_POLL_UPCALLS capability would require us to signal the ibis implementation that a new message might be available.

Lines 13-16 show the upcall method as it is implemented by this application. Again, we read a string from the message and print it. Notice there is no call to `finish` on this message. Whenever the upcall ends the message is automatically finished. It is possible to call `finish` though, signaling Ibis that the next message can now be received. This

```
1  public class HelloUpcall implements MessageUpcall {
2
3      PortType portType =
4          new PortType(PortType.COMMUNICATION_RELIABLE,
5                  PortType.SERIALIZATION_DATA, PortType.RECEIVE_AUTO_UPCALLS,
6                  PortType.CONNECTION_ONE_TO_ONE);
7
8      ...
9
10     /**
11      * Function called by Ibis to give us a newly arrived message.
12      */
13     public void upcall(ReadMessage message) throws IOException {
14         String s = message.readString();
15         System.out.println("Received string: " + s);
16     }
17
18     private void server(Ibis myIbis) throws IOException {
19         // Create a receive port, pass ourselves as the message upcall
20         // handler
21         ReceivePort receiver =
22             myIbis.createReceivePort(portType, "server", this);
23         // enable connections
24         receiver.enableConnections();
25         // enable upcalls
26         receiver.enableMessageUpcalls();
27     }
28 ...
```

Figure 5: Excerpt from application that uses upcalls

might be a good idea if a lot of calculations are done in the upcall method. If `finish` is not called, Ibis will not receive the next message until this upcall exits. Also, when you want to make calls into Ibis, like creating a new send message, setting up new connections, etc, you *must* finish the read message first, to prevent deadlocks in Ibis.

Finally, lines 18 to 27 show the server method of this application. One difference with the previous application can be seen on line 21, when the receive port is created. Now, we pass the object which will be called whenever a message is received to Ibis. In this case, this is simply ourselves (the third parameter with a `this` value). Also, we now need to no only enable connections to this message in line 24, but must also enable upcalls in line 26.

4.3 Registry

After examples of how to send and receive messages, we will now give an example of an application which uses the registry. This application, for which the source is listed in Figure 6, prints out whenever an Ibis joins the pool, an Ibis leaves the pool, or an Ibis dies.

Since this application does not actually send any messages, no port type is defined. Instead, only the `MEMBERSHIP_TOTALLY_ORDERED` capability is specified (lines 7-

```
1  package ibis.ipl.examples;
2
3  import ibis.ipl.*;
4
5  public class RegistryUpcalls implements RegistryEventHandler {
6
7      IbisCapabilities ibisCapabilities =
8          new IbisCapabilities(IbisCapabilities.MEMBERSHIP_TOTALLY_ORDERED);
9
10     // Methods of the registry event handler. We only implement the
11     // join/leave/died methods, as signals and elections are disabled
12
13     public void joined(IbisIdentifier joinedIbis) {
14         System.err.println("Got event from registry: " + joinedIbis
15                 + " joined pool");
16     }
17
18     public void died(IbisIdentifier corpse) {
19         System.err.println("Got event from registry: " + corpse + " died!");
20     }
21
22     public void left(IbisIdentifier leftIbis) {
23         System.err.println("Got event from registry: " + leftIbis + " left");
24     }
25
26     public void gotSignal(String signal) {
27         // NOTHING
28     }
29
30     public void electionResult(String electionName, IbisIdentifier winner) {
31         // NOTHING
32     }
33
34     public void poolClosed() {
35         // NOTHING
36     }
37
38     public void poolTerminated(IbisIdentifier source) {
39         // NOTHING
40     }
41
42     private void run() throws Exception {
43         // Create an ibis instance, pass ourselves as the event handler
44         Ibis ibis = IbisFactory.createIbis(ibisCapabilities, this);
45         ibis.registry().enableEvents();
46
47         // sleep for 30 seconds
48         Thread.sleep(30000);
49
50         // End ibis.
51         ibis.end();
52     }
53
54     public static void main(String args[]) {
55         try {
56             new RegistryUpcalls().run();
57         } catch (Exception e) {
58             e.printStackTrace(System.err);
59         }
60     }
61 }
```

Figure 6: Complete source of registry example application

8). This enables the membership management features of Ibis.

The `main` method of this application creates an object for this application, and calls the `run` method defined on lines 42-52. This method creates an Ibis on line 44. In this call, two things are passed. First, the capabilities of this Ibis, in this case only the single capability. Second, a `RegistryEventHandler` needs to be passed to automatically receive registry events such as joins and leaves of Ibis instances. We pass ourselves as the event handler. The class implements the `RegistryEventHandler` interface defined in the IPL. This interface consists of 7 methods. These methods are implemented on lines 13 to 40 in this example. Wherever a new Ibis joins the pool, the `joined` method is called by the registry. Likewise, when a Ibis leaves the pool, or when it crashes the `left` or `died` methods are called. In this example, all these events are handled by simply printing out the event that occurred.

The next two methods, `gotSignal` and `electionResult`, are not implemented here, since we have not enabled signals or elections. However, if we added signals and election to our capabilities, these methods would be called whenever a signal was received or a new election result was available.

The last two methods, `poolClosed` and `poolTerminated`, are also not implemented here, since we have not enabled the CLOSED_WORLD capability, not the TERMINATION capability.

This application uses upcalls to receive the events of the registry. However, it is also possible to poll the registry. By enabling a membership capability, but not passing an event handler, a user can call the `joinedIbisis`, `leftIbisis` and `diedIbisis` methods in the `Registry` class, which return any changes to the pool since the methods were last called. For an example of such an application, see the `RegistryDowncalls` example application.

4.4 Other Examples

The Ibis distribution also contains some other examples. These include a `OneToMany` example showing a simple broadcast application, and a `ManyToOne` example which resembles the HelloUpcall application, but with multiple clients.

The last application in the `example` directory is the `ClientServer` application. This application uses object serialization and multiple port types to implement a time server demo. It also shows the usage of a *ReceivePortIdentifier*. Instead of passing the Ibis and name of a receive port when connecting, a user can instead pass the unique identifier of a receiver port directly.

5 Further Reading

The Javadoc included in the `javadoc` directory has detailed information on all classes and their methods.

The Ibis web page `http://www.cs.vu.nl/ibis` lists all the documentation and software available for Ibis, including papers, and slides of presentations.

For detailed information on running an Ibis application see the User's Manual, available in the docs directory of the Ibis distribution.