Ibis Programmer's Manual

The Ibis Group

September 15, 2006

1 Introduction

Ibis is an efficient and flexible Java-based programming environment for Grid computing, in particular for distributed supercomputing applications. This manual describes how to write and run Ibis applications. It is available on-line at http://www.cs.vu.nl/ibis/progman. Ibis is described in several publications (see Section 8). 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 *docs/api* subdirectory of the Ibis installation provides documentation for each class and method (point your favorite browser to docs/api/index.html file of the Ibis installation). The Ibis API is also available on-line at http://www.cs.vu.nl/ibis/api. 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.

We also built several systems on top of Ibis. Section 5 gives an overview of the Satin divide-and-conquer system, and Section 7 discusses GMI (Group Method Invocation), a flexible group communication system. We also built a RMI (Remote Method Invocation) implementation on top of Ibis. RMI documentation can be found at http://java.sun.com/j2se/1.4.2/docs/guide/rmi. Section 6 briefly discusses the Ibis RMI implementation.

2 Some Ibis concepts

2.1 The Ibis Portability Layer

An important part of the Ibis system consists of the Ibis Portability Layer (IPL). The 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 properties that it requires, and the Ibis system selects the best available Ibis implementation that meets these requirements.

2.2 An Ibis Instance

A loaded Ibis 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 *ResizeHandler*. This ResizeHandler 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 communicate between send and receive ports. In general a connection can be between multiple send ports and multiple receive ports, but the user may specify that a connection will have only a single send or receive port, allowing Ibis to choose a more efficient implementation. A connection is always *unidirectional*; reverse connections are conceptually totally independent.

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 requests new send and receive ports from such a port type, and requests that a connection is set up between these ports.

To send a message, the Ibis application requests a new write message from a send port, puts data in this write message using the provided methods, and invokes the `finish()` method to send the message.

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 3.2.

2.4 Port Types

Send and receive ports are *typed* by means of a *port type*. A port type is defined and configured with properties. Only ports of the same type can be connected. Port type properties 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.

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. In Ibis, we call this mechanism *Sun serialization*. Ibis also has its own mechanism, which is completely compatible (with regard to its interface) with Sun serialization, but more efficient. We call this mechanism *Ibis serialization*.

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 client and a server. As this is a toy application, the server will have to compute the length of a string. The client will send the string, and receive the result. 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`. We recommend that you 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, but this tends to result in a list of 10 or more `imports`.

3.2 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.Ibis` class must be used. The specification of this method is:

```
static Ibis createIbis(StaticProperties props,
                       ResizeHandler h)
        throws IbisException;
```

There may be several Ibis implementations available, and the system selects the best one for you, based on some user-specified requirements. These requirements tell the system what features must be supported by the selected Ibis. They are summarized in an object of the `ibis.ipl.StaticProperties` class, as discussed in Section 3.3

The second parameter of `createIbis()` specifies an upcall handler with `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 `ResizeHandler` is used, its `joined()` upcall is called for every Ibis that joins the run, including the Ibis being created itself. Upcalls to the `ResizeHandler` must be explicitly enabled by invoking the `enableResizeUpcalls()` method of the Ibis just created. This ensures that the `ResizeHandler` 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 3.4.1.

Now back to our example. In Secion 3.3 we will discuss the creation of a variable `props` describing the static properties required. For the moment we will just assume its existence, and create an Ibis instance as follows:

```
Ibis ibis = null;
try {
    ibis = Ibis.createIbis(props, null);
} catch(NoMatchingIbisException e) {
    System.err.println("Could not find a matching Ibis");
    ...
}
```

Note that the properties can be so specific that no matching Ibis can be found, and therefore `createIbis()` may throw an exception indicating this.

3.3 StaticProperties

Below is a snippet from the RPC example, constructing the static properties as required:

```
StaticProperties props = new StaticProperties();
props.add("communication", "OneToOne, Reliable, " +
                           "AutoUpcalls, ExplicitReceipt");
props.add("serialization", "object");
props.add("worldmodel", "closed");
```

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, and a synchronization takes place before `createIbis()` returns (in the example we have a client and a server).

A complete list of property values is given below. The possible property values of the `communication` property are (capitals are not significant):

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

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

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

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

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.

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.

AutoUpcalls 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.

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

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

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

ConnectionUpcalls 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.

The possible `serialization` properties are:

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

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

Object Some sort of object serialization is supported. An Ibis implementation will, of course, specify what kind of object serialization it supports. The "Object" property allows a user to just ask for object serialization, and not care if it is Ibis or Sun serialization.

Ibis Ibis serialization is supported. This is the fastest object serialization supported in Ibis. Its drawback is that it requires identical Java implementations on sender and receiver side; it also 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).

Sun Sun serialization (through `java.io.ObjectOutputStream/InputStream`) is supported.

The possible `worldmodel` properties are:

Open Ibises can join the run at any time during the run.

Closed The number of nodes involved in the run is known in advance and available from `ibis.util.PoolInfo` (see Section 3.8.1).

If a specific implementation of Ibis is required, that can be dealt with too. There is a property called `name`, which can be used to supply a nickname for the Ibis implementation that is required. The currently known nicknames are:

tcp This is an Ibis implementation on top of TCP sockets. It is currently probably the most stable Ibis around, and it supports almost all properties.

panda This is a message passing Ibis implementation on top of the Panda communication layer. It only supports the "closed" worldmodel, but on our system it gives much higher throughput and lower latencies because it runs on Myrinet instead of our (100Mbitps) Ethernet on which the TCP version runs.

net This is the most flexible Ibis implementation, and is becoming quite stable now. It can use multiple networks simultaneously.

nio This is an Ibis implementation on top of Java NIO.

Alternatively, you can specify the fully qualified name of an implementation of an `ibis.ipl.Ibis` subclass. You can use this to indicate that you want to use your own ibis implementation, for which no nickname exists.

A user running an Ibis application can override a property, or make it more specific. This is done by means of Java system properties, which can be set by means of a command line option or with the `System.setProperty` method. For instance, Sun and IBM JVMs support options of the form -D*property*=*value*. The system property name is that of the Ibis static property, prefixed with "`ibis.`". So, adding `-Dibis.serialization=ibis` to the command line will cause `createIbis()` to look for an Ibis implementation that supports Ibis serialization instead of any object serialization.

3.4 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.

3.4.1 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);
```

In our example, one instance of the program is the server and all other instances are clients. Of course, the client and the server can also be different programs altogether.

The `ResizeHandler`, as discussed in Section 3.2, can be used to keep track of the number of Ibis instances currently involved in the run.

3.4.2 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 properties of a port type are, like the required properties of an Ibis implementation, specified by a `StaticProperties` object. A port type is identified by its name, together with these properties. The `Ibis` class contains a method to create a port type, specified as follows:

```
PortType createPortType(String name,
                        StaticProperties portprops)
        throws IbisException, java.io.IOException;
```

For our RPC example program, we would create a port type with properties as discussed in Section 3.2:

```
StaticProperties portprops = new StaticProperties();
portprops.add("communication",
              "OneToOne, Reliable, " +
                  "AutoUpcalls, ExplicitReceipt");
portprops.add("serialization", "object");
PortType porttype = null;
try {
    porttype = ibis.createPortType("RPC port", portprops);
} catch(Exception e) {
    ...
}
```

In general, the port properties should be a subset of the properties specified when creating the Ibis instance. If a property is specified that was not specified when creating the Ibis instance, this may result in an `IbisException`, depending on the strictness of the Ibis implementation. An `IbisException` will also result when the same name is already used for a port type with different properties. Port types are registered with a name server (this will be discussed in Section 4). If communication with this name server fails, a `java.io.IOException` is thrown.

3.4.3 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.Upcall` 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 3.6. 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 `enableUpcalls()` method. Again, incoming messages are kept pending until upcalls are enabled.

3.4.4 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 (ReceivePortIdentifier r) 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 `ibis.ipl.ReceivePortIdentifier` to set up the connection. This identifier can be obtained through the Ibis registry, by means of the `lookupReceivePort()` method, which has the following specification:

```
ReceivePortIdentifier lookupReceivePort(String name)
        throws IOException;
```

This method blocks until a receive port with the specified name is found. Our RPC server would set up the following connection:

```
try {
    ReceivePortIdentifier client
            = registry.lookupReceivePort("client");
    serverSender.connect(client);
} catch(IOException e) {
    ...
}
```

Our RPC client would set up the following connection:

```
try {
    ReceivePortIdentifier server
            = registry.lookupReceivePort("server");
    clientSender.connect(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 `OneToMany` communication property). Also, multiple send ports can set up connections to the same receive port (if the port type supports the `ManyToOne` communication property).

3.5 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 3.3 for the corresponding properties.

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 `SendPortConnectUpcall` 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.

3.6 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 property 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

property does not allow use of the `writeObject()` method, but does allow the use of all other write methods. The `byte` serialization property 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 needed, 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 = (String) m.readObject();
    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;
}
```

3.7 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.

3.8 Ibis utilities

The `ibis.util` package contains several utilities that may be useful for Ibis applications. We will discuss some of them here.

3.8.1 PoolInfo

The `ibis.util.PoolInfo` utility provides methods for finding out information about the nodes involved in a closed-world run, such as:

- the total number of hosts involved in the run.
- the rank number of the current host in the pool.
- the host names of the hosts involved in the run.
- the `InetAddress` of the hosts involved in the run.

A `PoolInfo` instance is created with its static method `createPoolInfo()`. It depends on the following system properties:

ibis.pool.total_hosts This system property must be present for closed-world runs. It indicates the total number of hosts involved in the run.

ibis.pool.host_names If present, this system property contains the list of host names involved in the run. If not present, `createPoolInfo()` instantiates a `PoolInfoClient` that will collect this information during startup. A `PoolInfoClient` uses a `PoolInfoServer` to collect the required information. This `PoolInfoServer` usually is started by the Ibis nameserver (see Section 4.1).

ibis.pool.host_number If present, this system property indicates the rank number of the current host. If not present, the system will provide a rank number.

3.8.2 Other utilities

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.

3.9 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 `ManyToOne` as well as the `OneToMany` communication property. 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 Nameserver

Most Ibis implementations depend on a nameserver for providing information about a particular run, such as finding Ibis instances participating in the run, finding or registering receive ports, et cetera. The Ibis nameserver 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 nameserver, using this identifier, so that the nameserver can determine to which Ibis run this Ibis instance belongs. The nameserver 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 nameserver location is a machine that also participates in the run itself, Ibis will automatically try to start a nameserver. How you can specify this is explained in the next section. If you want to run the nameserver on a seperate host, one that is not behind a firewall, for instance, you have to start the nameserver by hand.

The Ibis nameserver is started with the `ibis-nameserver` script which lives in the Ibis `bin` directory. Before starting an Ibis application, you need to have a nameserver running on a machine that is accessible from all nodes participating in the Ibis run. The nameserver 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 a command line option to the `ibis-nameserver` script. This script recognizes the following options:

`-single` specifies that the nameserver should only serve a single Ibis run and then exit.

`-port` *portno* specifies the port number of the nameserver socket. A port number must be between 0 and 65535, inclusive. Usually, port numbers below 1024 are reserved for other purposes.

`-poolport` *poolportno* specifies the port number of the `PoolInfoServer` mentioned in Section 3.8.1.

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

The most important *ibis-run-flags* are explained below:

`-nhosts` *nhosts* specifies the total number of Ibis instances involved in this run.

`-hostno` *hostno* specifies the rank number of this Ibis instance within this run, so its range is $0...nhosts - 1$.

`-key` *id* specifies the identifier used to identify the run to the nameserver.

`-ns` *nameserverhost* specifies the hostname of the machine where the Ibis nameserver runs. If the hostname refers to a machine that also participates in the run, Ibis will start the nameserver itself.

`-ns-port` *port* specifies the port number on which the nameserver is listening.

`-?` gives a description of all options.

javaflags are any flags that need to be passed on to java.

class specifies the application class name.

params specifies the optional application parameters.

The `ibis-run` script uses these parameters to set the following system properties (see also Section 3.8.1):

`ibis.pool.host_number` the rank number of this Ibis instance.

`ibis.pool.total_hosts` the total number of Ibis instances.

`ibis.name_server.key` identifies the run to the nameserver.

`ibis.name_server.port` the nameserver port.

`ibis.name_server.host` the nameserver hostname.

In addition, the following system properties can be specified (as *javaflags*):

ibis.pool.cluster specifies a cluster name. If not specified, "unknown" is used. See Section 5.3 for a possible need for cluster names.

ibis.pool.server.port specifies the port number on which the `PoolInfoServer` is listening.

The Ibis distribution also provides *grun*, which is a tool for running Ibis applications on a Globus-based grid. See the `grun/doc` subdirectory for more information.

4.3 Running the example

In order to run the example, we first have to compile it. Please go to the *ibis-example* directory and type:

```
$ ant
```

After a couple of seconds, the example should be compiled. You can run the example both with and without the `ibis-run` script. Because we run the example on a single machine, we do not need to start a seperate nameserver. To run the application, we first need to start two shells. Then, in both shells type:

```
$ $IBIS_HOME/bin/ibis-run \
    -nhosts 2 -ns localhost Example
```

Now, it should run and print:

```
Test succeeded!
```

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

```
$ java \
    -cp $IBIS_HOME/lib/ibis.jar:$IBIS_HOME/3rdparty/log4j-1.2.9.jar:build \
    -Dibis.pool.total_hosts=2 -Dibis.name_server.host=localhost \
    -Dibis.name_server.key=bla \
    Example
```

The `ibis.name_server.key` value can be any random string. It identifies your run. This is done because one nameserver can serve multiple runs. For this test, we need to provide Ibis with the total number of CPUs in the run, because it is a closed-world test. Ibis waits until both processors have joined the computation. The `ibis.name_server.host` property should be set to the machine you run the nameserver on. In this case, we use `localhost`. Because we also run the application on `localhost`, Ibis will automatically start a nameserver. If you provide a hostname where the Ibis application does not run, you will have to start a nameserver yourself.

5 The Satin Divide-and-Conquer System

Satin is a parallel programming environment for divide-and-conquer parallelization, and master-worker parallelization. Satin extends Java with two simple primitives for divide-and-conquer programming: `spawn` and `sync`. The Satin byte-code rewriter and runtime system cooperate to implement these primitives efficiently on top of the IPL.

5.1 Satin jobs

To use Satin, the programmer must label one or more methods in his class as Satin jobs. This is done by defining an interface that extends the Satin interface ibis.satin.Spawnable. For example:

```
interface Searcher extends ibis.satin.Spawnable {
   public int search(int a[], int from, int to, int val);
}
```

All methods that implement Searcher.search() will be Satin jobs, they are marked as spawnable. In general such a marker interface may contain an arbitrary number of methods. When a Satin program is compiled with the Satin compiler, methods marked as spawnable may be executed in parallel. A class that has spawnable methods must extend the special class ibis.satin.SatinObject. The result of a Satin job can only be used after the invocation of the sync method, which lives in ibis.satin.SatinObject. The sync method is guaranteed to only terminate after the earlier job invocations have terminated. Satin imposes one restriction on the type of parameters and return type of a Satin job: they must be of a basic type or must be serializable. Since a SatinObject is serializable, all its subclasses are serializable as well. This restriction ensures that a Satin job can be stored and moved from one processor to another. Note that this means that all fields of a Satin object must be either serializable or transient.

As an example, Figure 1 shows an implementation of the search method. It just looks at a range of elements of the passed array, and tries if one of the elements equals val. If so, it returns the index of the element. If no matching element was found, -1 is returned. Now, we can invoke search as is shown in the main method of Figure 1. In this case, we call search twice, once for the first half of the array, and once for the second half. If we compile the program with javac, the two search invocations will be done sequentially. The sync method in SatinObject does nothing. So we can run the program normally on any JVM. If we modify the class files using the Satin bytecode rewriter, however, the program will be converted to a parallel program. The two calls to search in the main class of Figure 1 will be executed in parallel if two JVMs are available. To run the code on more JVMs, the array can be split into more chunks. In general, an arbitrary number of invocations to Satin job methods may be done.

It is also allowed to recursively invoke job methods from job methods, as is shown in Figure 2. Here the search is recursively divided into smaller problems until a problem remains that is trivially handled. This model of programming is called divide-and-conquer. Satin has special grid-aware load-balancing algorithms built-in to run such programs efficiently on the grid, on any number of machines. This way, the entire search can be done in parallel on a large number of machines. The example shows how easy it is to write a real parallel grid application using Satin.

```
import ibis.satin.SatinObject;

class SearchImpl1 extends SatinObject implements Searcher {
   public int search(int a[], int from, int to, int val) {
      for(int i = from; i < to; i++) {
         if (a[i] == val) return i;
      }
      return -1;
   }

   public static void main(String[] args) {
      SearchImpl1 s = new SearchImpl1();
      int a[] = new int[200];

      // Fill the array with random values between 0 and 100.
      for(int i=0; i<200; i++) {
         a[i] = (int) (Math.random() * 100);
      }

      // Search for 42 in two sub-domains of the array.
      // Because the search method is marked as spawnable,
      // Satin can run these methods in parallel.
      int res1 = s.search(a, 0, 100, 42);
      int res2 = s.search(a, 100, 200, 42);

      // Wait for results of the two invocations above.
      s.sync();

      // Now compute an overall result.
      int res = (res1 >= 0) ? res1 : res2;

      if(res >= 0) {
         System.out.println("found at pos: " + res);
      } else {
         System.out.println("element not found");
      }
   }
}
```

Figure 1: Parallel search with Satin.

5.2 Exceptions and abort

A Satin job may throw exceptions in the usual way. This can be used to rapidly terminate a large set of jobs, e.g., when a search result was found. Terminating jobs that are no longer useful can be done with the abort method from the SatinObject class. Satin has its own exception type: `ibis.satin.Inlet`. By extending the `Inlet` class, the programmer can inhibit the generation of a stack

```
import ibis.satin.SatinObject;

class SearchImpl2 extends SatinObject implements Searcher {
   public int search(int a[], int from, int to, int val) {
      if (from == to) { // The complete array has been searched.
          return -1;    // The element was not found.
      }
      if (to - from == 1) { // Only one element left.
          return (a[from] == val) ? from : -1; // It might be the one.
      }

      // Now, split the array in two parts and search them in parallel.
      int mid = (from + to) / 2;
      int res1 = search(a, from, mid, val);
      int res2 = search(a, mid, to, val);
      sync();
      return (res1 >= 0) ? res1 : res2;
   }

   // main method as in previous figure
}
```

Figure 2: Divide-and-conquer parallel search with Satin.

trace, which is an expensive operation. The stack trace is usually not useful in the Satin context, because the exception is not an error, it is used to steer the search. Extending `Inlet` is optional, regular exceptions can also be used. Figures 3 and 4 change the search algorithm we used in the previous examples to use exceptions.

Note that in this version the search method doesn't return a value. It will throw a SearchResultFound object containing the result if the element was found, or it will terminate without throwing an exception if no result is found. This is also the reason the search method implements the Searcher3 interface instead of the Seacher interface. The declaration of the exception in the throws clause and the return type are the only differences. At the top level this exception is caught, and the other search jobs can then be terminated. They have become useless, because the element has already been found in the array. The parallel search can thus be stopped. We call the catch block in the main method that handles the result of the search an inlet. The way of searching in Figures 3 and 4 is called speculative parallelism: the search speculatively starts on both two parts of the array simultaneously, even though we know from the start that a part of the search may be terminated as soon as the element is found. We might thus do more work than a sequential search. On the other hand, we might also do less work than the sequential version, for instance if the element is in the beginning of the second part of the array. The inlet in main contains a return statement. This must be there, because the inlet runs in a new thread.

```
import ibis.satin.SatinObject;
import ibis.satin.Inlet;

class SearchResultFound extends Inlet {
   int pos;

   SearchResultFound(int pos) {
      this.pos = pos;
   }
}

interface Searcher3 extends ibis.satin.Spawnable {
   public void search(int a[], int from, int to, int val)
      throws SearchResultFound;
}

class SearchImpl3 extends SatinObject implements Searcher3 {
   public void search(int a[], int from, int to, int val)
         throws SearchResultFound {

      if (from == to) { // The complete array has been searched.
         return;        // The element was not found.
      }
      if (to - from == 1) {    // Only one element left.
         if (a[from] == val) { // Found it!
            throw new SearchResultFound(from);
         } else {
            return; // The element was not found.
         }
      }

      // Now, split the array in two parts and search them in parallel.
      int mid = (from + to) / 2;
      search(a, from, mid, val);
      search(a, mid, to, val);
      sync();
   }

   ...
```

Figure 3: Speculative parallel search with Satin.

The original main thread is still blocked in the sync statement. When the inlet returns, the main thread is unblocked, because both search jobs have finished: one threw an exception, the other has been aborted.

```
   ...

   public static void main(String[] args) {
      SearchImpl3 s = new SearchImpl3();
      int a[] = new int[200];
      int res = -1;

      // Fill the array with random values between 0 and 100.
      for(int i=0; i<200; i++) {
         a[i] = (int) (Math.random() * 100);
      }

      // Search for 42 in two sub-domains of the array.
      // Because the search method is marked as spawnable,
      // Satin can run these methods in parallel.
      try {
         s.search(a, 0, 100, 42);
         s.search(a, 100, 200, 42);

         // Wait for results of the invocations above.
         s.sync();
      } catch (SearchResultFound x) {
         // We come here only if one of the two jobs found a result.
         s.abort(); // kill the other job that might still be running.
         res = x.pos;
         return; // return needed because inlet is handled in separate thread.
      }

      if(res >= 0) {
         System.out.println("found at pos: " + res);
      } else {
         System.out.println("element not found");
      }
   }
}
```

Figure 4: Speculative parallel search with Satin.

5.3 Satin job scheduling

Satin offers a choice of three different job scheduling strategies:

Random work-stealing. When looking for work, each Satin instance first examines its own job queue. When this job queue is empty, it randomly selects another Satin instance and takes the first job on its job queue. If that job queue is empty, it randomly selects another Satin instance, et cetera, et cetera. This strategy is the default, and has in the past been proven to be optimal, although that might seem counter-intuitive.

Cluster-aware random work-stealing. The word "cluster" refers to a group of computers that are connected to each other by means of a fast local network. In turn, clusters can be connected to each other, but usually the network between clusters is much slower, and/or has a much higher latency. The cluster-aware random work-stealing strategy is very similar to random work-stealing, except that each participant first tries to steal jobs from other participants in its own cluster (thus using the fast local network). It will only go to participants from other clusters if no participant on the same cluster has work. To determine to which cluster a participant belongs, the cluster name is used. As described in Section 4.2, the `ibis.pool.cluster` system property can be used to specify a cluster name to the Ibis instance.

Master-worker This strategy is suitable for applications where all Satin jobs are generated on a single participant, the "master". All other participants are "workers", they obtain jobs from the master and execute them.

Making sure that all participants always can find work to do may need some tuning. Jobs must not be too small, because otherwise the mechanism for obtaining jobs, which may involve network traffic, is too expensive. On the other hand, there must be enough jobs to keep everybody busy. The `SatinObject` class has a boolean method `needMoreJobs()`, which indicates whether it would be useful to generate more jobs.

5.4 Other SatinObject methods

The `pause()` method pauses Satin's operation. When the application contains a large sequential part, this method can be called to temporarily pause Satin's internal load distribution strategies to avoid communication overhead during the execution of sequential code. To resume Satin's operation, the `resume()` method must be used.

5.5 Satin program arguments

The Satin system accepts the following parameters, which are passed to java as system property values.

`-Dsatin.closed` Only use the initial set of hosts for the computation; do not allow further hosts to join the computation later on.

`-Dsatin.stats` Display some statistics at the end of the Satin run. This is the default.

`-Dsatin.stats=false` Don't display statistics.

`-Dsatin.detailedStats` Display detailed statistics for every member at the end of the Satin run.

`-Dsatin.alg=`*algorithm* Specify the load-balancing algorithm to use. The possible values for *algorithm* are: RS for random work-stealing, CRS for cluster-aware random-work stealing, and MW for master-worker.

5.6 Satin Shared Objects

The divide-and-conquer model operates by subdividing the problem into subproblems (subtasks) and solving them recursively. The only way of sharing data between tasks is by passing parameters and returning results. Therefore, a task can share data with its subtasks and the other way round, but the subtasks cannot share data with each other. Therefore, we have extended the model, and in the process renamed it "divide-and-share".

In the divide-and-share model, tasks can share data using *shared objects*. Updates performed on a shared object are visible to all tasks. Operations on shared objects are executed *atomically*. Satin guarantees that shared object operations do not run concurrently with each other. Satin also guarantees that the shared object operations do not run concurrently with divide-and-conquer tasks. An operation performed by a task becomes visible to other tasks only when the system reaches a so-called *safe point*: when a task is creating (spawning) subtasks, when a task is waiting for its subtasks to finish, or when a task completes. Tasks can also explicitly poll for shared object updates. This makes the model clean and easy to use, as the programmer does not need to use locks and semaphores to synchronize access to shared data.

Shared objects are replicated on every processor taking part in the computation. Replication is implemented using an update protocol with function shipping: *write methods*, that is, methods that modify the state of the object, are forwarded to all processors which apply them on their local replicas. *Read methods*, that is, methods that do not change the state of the objects, are executed locally.

A shared object type must be marked as such by extending the special class `ibis.satin.SharedObject`. This class exports a single method

```
public void exportObject();
```

which may optionally be invoked to inform Satin that it might be useful to broadcast the object to all participants in the run.

Write methods must be marked as such, by specifying them in an interface that extends the marker interface `ibis.satin.WriteMethodsInterface`. Note that it is up to the user to specify which methods are write methods. Here is a small example:

```
interface MinInterface extends ibis.satin.WriteMethodsInterface {
    public void set(int val);
}

final class Min extends ibis.satin.SharedObject
        implements MinInterface {
    private int val = Integer.MAX_VALUE;
    public void set(int newVal) {
```

```
            if (newVal< val) val = newVal;
        }
        public int get() { return val; }
    }
```

Our shared object model provides a relaxed consistency model called *guard consistency*. Under guard consistency, the user can define the application consistency requirements using *guard functions*. Guard functions are associated with divide-and-conquer tasks. Conceptually, a guard function is executed before each divide-and-conquer task. A guard checks the state of the shared objects accessed by the task and returns *true* if those objects are in a correct state, or *false* otherwise. Satin allows replicas to become inconsistent as long as guards are satisfied: the updates are propagated to remote replicas on a best effort basis. Satin does not guarantee that the updates will not be lost or duplicated. Updates may be applied in different order on different replicas. When a guard is unsatisfied, Satin invalidates the local replica of a shared object and fetches a consistent replica from another processor.

For each spawnable method, the programmer may define a boolean guard method which specifies what the state of shared object parameters should be before actually executing the spawned method. Such a guard function must be defined in the same class as the spawnable method it guards. The name of the guard function is `guard_`*spawnable_function*. It must have exactly the same parameter list as the spawnable method and return a boolean value. Therefore, it will have access to exactly the same shared objects and it can check their consistency. It will also have access to other parameters of the Satin task on which the consistency state of the objects may depend. Satin takes care of invoking this method at the right moment. Here is a small example:

```
/* A spawnable method. */
public Result compute(int iteration, Data data) {
    ...
}

/* Guard: make sure that data represents the right iteration. */
public boolean guard_compute(int iteration, Data data) {
    return data.iteration == iteration-1;
}
```

5.7 Compiling and running a Satin program

Before running a Satin application it must be compiled. Using *ant*, this is quite easy. Here is a build.xml file for our example program:

```
<project
    name="Search"
    default="build"
    basedir=".">
```

```
    <description>
    Satin application build.
    </description>

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

    <property name="satin-classes" value="Search"/>

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

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

Again, we assume that the environment variable `IBIS_HOME` reflects the location of your Ibis installation. Also note that the Satin bytecode rewriter needs to know the classes in your application that must be rewritten. Those classes are: the main class, any class that implements a shared object, and all classes that invoke spawns or syncs. The names of these classes are specified in the `satin-classes` property, as a comma-separated list.

Invoking *ant clean compile* compiles a sequential version of the application, leaving the class files in a directory called `build`.

Invoking *ant clean build* compiles the application for parallel runs.

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. To process a Satin application, *ibisc* needs to know which classes to process for Satin. This can be accomplished by passing flags to *ibisc*:

`-satin` *comma-separated list of class names*

where the comma-separated list of class names consists of the main class, any class that implements a shared object, and all classes that invoke spawns or syncs.

Running a Satin program is very much like running an Ibis application. See Section 4 for details.

6 Ibis RMI

Java applications typically consist of one or more threads that manipulate a collection of objects by invoking methods on these objects. Figure 5 shows an example, where a single application thread has a reference to some interface which represents an object. The thread and object are located in the same Java Virtual Machine (JVM), and the thread can use normal method invocations on the interface to manipulate the state of the object.

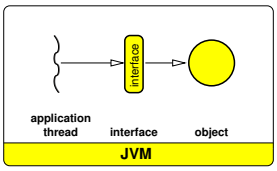

Figure 5: A normal invocation.

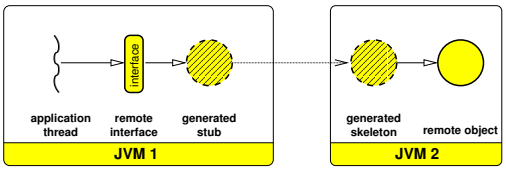

Figure 6: A remote invocation with RMI.

To turn the example of Figure 5 into a distributed RMI invocation, some small modifications must be made to the program. The interface must be turned into a remote interface by extending java.rmi.Remote, and the object must be turned into a remote object by extending java.rmi.UnicastRemoteObject. The rmic compiler, which is part of the Java Developer Kit (JDK), can then generate the required communication code. This code consist of two objects, a 'stub' and a 'skeleton', as shown in Figure 6.

The stub object implements the application interface, and contains code to forward any method invocations it receives to a skeleton object on another JVM. The skeleton object contains code to receive these invocations, and perform them on the object. It then sends the results back to the stub, which returns them to the waiting application thread. Although RMI is not completely transparent, only small modifications to the application are required. Furthermore, the programmer does not have to write any communication code (this is generated by rmic), making RMI easy to use. Unfortunately, the way in which method invocations are handled in RMI is fixed. After the stub forwards

the invocation to the skeleton, it waits for a reply message before continuing. The skeleton must therefore always send a reply back to the stub (even if the method has no result. Furthermore, a stub in RMI always serves as a 'remote reference' to a single object, which can not be changed once the stub has been created.

There is very little difference between the usage of Sun RMI and Ibis RMI. The programs are exactly the same, you only have to compile them with Ibis `rmic` instead of the Sun `rmic`.

6.1 Compiling and running an Ibis RMI program

Before running an Ibis RMI application it must be compiled. Using *ant*, this is quite easy. Here is an example build.xml file:

```
<project
    name="Project name"
    default="build"
    basedir=".">

    <description>
    Ibis RMI 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-rmi-app.xml"/>
</project>
```

Again, we assume that the environment variable `IBIS_HOME` reflects the location of your Ibis installation.

Invoking *ant build-sun* compiles a standard Java RMI version of the application, leaving the class files in a directory called `build`. Invoking *ant build* compiles the application for Ibis RMI. Running an Ibis RMI program is very much like running an Ibis application.

To test this for yourself, it is best to start with an example, say `$IBIS_HOME/apps/rmi/tsp`. If you want to test your own application, it is easiest to just copy the `build.xml` file from `$IBIS_HOME/apps/rmi/tsp`, and adapt it for your application. To compile the TSP example, please type

```
$ ant clean build-sun
```

Now, this example application will be compiled with Sun RMI. Running the TSP example with Sun RMI can be done as follows. Run the command

```
$ java \
    -cp $IBIS_HOME/lib/ibis.jar:$IBIS_HOME/3rdparty/log4j-1.2.9.jar:build \
    -Dibis.pool.total_hosts=2 -Dibis.pool.server.host=localhost \
    Server table_15.1
```

in two seperate shells. The application should now run "in parallel" on the local machine. Alternatively, you can use the *ibis-run* script:

```
$ $IBIS_HOME/bin/ibis-run -nhosts 2 -hostno 0 Server table_15.1
```

in the first shell, and

```
$ $IBIS_HOME/bin/ibis-run -nhosts 2 -hostno 1 Server table_15.1
```

in the second.

The TSP application uses the `PoolInfo` class that comes with Ibis. This utility class can be used both with Sun RMI and Ibis RMI. This class is there for convenience, it provides some methods to retrieve the number of processors in the parallel run and the ranks of the participating processors. Because this class comes with Ibis, `ibis.jar` has to be in the classpath, even when running with Sun RMI. The `PoolInfo` class needs the `ibis.pool.total_hosts` and `ibis.pool.server.host` properties in order to be able to assign ranks to processors. The pool server is started automatically when you start an ibis nameserver with the `ibis-namesever` script (in the case of this example, this is done automatically).

Now, we can run the same application with Ibis RMI as follows. Remember that you first have to recompile it with the Ibis rmic:

```
$ ant clean build
```

Now we can run it by typing the following command in two seperate shells:

```
$ java \
    -cp $IBIS_HOME/lib/ibis.jar:$IBIS_HOME/3rdparty/log4j-1.2.9.jar:build \
    -Dibis.pool.total_hosts=2 -Dibis.pool.server.host=localhost \
    -Dibis.name_server.host=localhost -Dibis.name_server.key=bla \
    Server table_15.1
```

This should produce the same result as the Sun RMI test. If you want to run the application with the `ibis-run` script, you can use the same commandline as with the Sun RMI test.

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. To process an RMI application, *ibisc* needs to know that it has to do so. This can be accomplished by passing either the `-rmi` or the `-rmi-java2ibis` flag to *ibisc*. The latter instructs *ibisc* to not only use the Ibis rmic, but also to change all references to `java.rmi` into references to `ibis.rmi`.

7 The GMI (Group Method Invocation) System

For many parallel and distributed applications, the simple synchronous unicast communication model (one-to-one communication with a reply) offered by RMI is inadequate. Applications often require more different, more complex forms of communication, such as asynchronous unicast (one-to-one communication without a reply), broadcast (one-to-all communication), multicast (one-to-many communication), or data reduction operations (where data must be collected from multiple machines). Although these alternative forms of communication can be implemented using RMI, this is often complex and inefficient. For example, a simple way to implement multicast is to perform multiple RMI calls, one after the other. Unfortunately, this is very inefficient, since each RMI must wait until the previous RMI has finished completely, including waiting for the (unused) result to be returned. More efficient implementations use threads to perform multiple RMIs simultaneously or create distributed multicast trees which use multiple machines to forward calls. These implementations are complex, however, and often suffer from performance problems caused by the synchronous nature of RMI. In Ibis, we offer a new programming model called GMI (Group Method Invocation). GMI is an extension of RMI designed to be flexible enough to express the complex forms of communication needed by many parallel applications. Nevertheless, GMI is as easy to use as RMI, hiding the complex details of the communication in its implementation. The flexible model of GMI also allows its implementation to use efficient communication algorithms for multicast, data reduction, etc. The GMI model will be explained in detail below.

7.1 The GMI model

The GMI model generalizes the RMI model in three ways. First, it introduces the notion of a 'group', a set of objects which all implement the same interface. The objects in a group may be distributed over a number of JVMs. A group can be addressed using a single 'group reference'. An example is shown in Figure 7, where an application thread uses a single group reference to address a group of two objects (on different JVMs).

Like RMI, GMI uses compiler generated stub and skeleton objects which implement the necessary communication code. The application programmer does not have to write any communication code. Unlike RMI, however, GMI does not only generate code for synchronous unicast communication. Instead, many different types of communication code are generated, both for sending of method invocations and for returning of results. This is the second generalization introduced by GMI. Finally, through a simple API, the programmer can configure at run time which type of communication should be used to handle each individual method of a group reference. This API will be explained in more detail in the next section. Different ways of forwarding the invocation and handling the results can be combined, giving a rich variety of communication mechanisms. By configuring the methods at run time, the communication

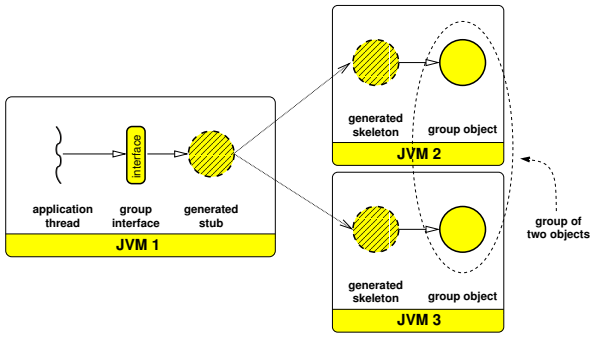

Figure 7: A group invocation with GMI.

behavior of the application can easily be adapted to changing requirements.

The following types of method invocation forwarding are currently supported by GMI:

- single invocation: The method invocation is forwarded to a single object of the group, identified via a rank.
- group invocation: The invocation is forwarded to every object in the group.
- personalized group invocation: The invocation is forwarded to every object in the group, while the parameters are personalized for each destination using a user-defined method.
- combined invocation: Multiple application threads (possibly on multiple JVMs) invoke the same method on the same group. These invocations are combined into a single invocation using a user-defined method. This single invocation is forwarded to the group using one of the three other forwarding schemes.

The following types of method result handling are currently supported by GMI:

- discard results: No results are returned at all (including exceptions).
- return one result: A single result is returned, preselected via a rank if neccesary.
- forward results: All results are returned, but they are forwarded to a user-defined object rather than being returned to the invoking thread.
- combine results: Combine all results into a single one using a user-defined method. The combined result is returned to the invoker.

- personalize result: A result produced by one of the other result handling schemes is personalized using a user-defined method before being returned to each of the invokers (this is useful when a combined invocation is used).

The four different forwarding schemes and five different result handling schemes can be combined orthogonally, resulting in a wide variety of useful communication patterns.

7.2 Hello world in GMI

We will now show a step by step example of how a GMI application can be written. The first step is to create an interface which will define the methods which can be invoked on the group of objects. Like in RMI, we use a special 'marker interface' to 'mark' group interfaces. Any interface extending ibis.gmi.GroupInterface will be recognized by the Ibis compiler as being a group interface. The Ibis compiler will then generate a stub object which contains the necessary communication code.

```
interface Example extends ibis.gmi.GroupInterface {
   public void put(String message);
   public String get();
}
```

In the example above, the 'Example' interface is turned into a group interface by extending ibis.gmi.GroupInterface. It defines just two methods, put, which can be used to store a string, and get which can be used to retrieve a stored string. After creating the group interface, an implementation of this interface is be created. This implementation object must implement the Example interface, and extend the ibis.gmi.GroupMember object, which contains some basic functionality needed to be part of a group (this is similar to the UnicastRemoteObject used in RMI).

In this implementation, shown in Figure 8, the put method stores the string it receives in the object, from where it can be retrieved using the get method. The standard synchronization primitives synchronized, wait, and notify are used to prevent the get from returning before the string is available. Next, we will create an simple example application, BroadcastExample, which will use the group object. The source code is shown in Figure 9. This application is parallel; it is designed to be started simultaneously on multiple JVMs.

The main method of the application starts by invoking Group.size and Group.rank, two utility methods of GMI which can be used to find out how many JVMs are available (size), and what number is assigned to the current JVM (rank). The JVM with rank 0 then creates a new group using Group.create. This group will have the name ExampleGroup, use a group interface of the type Example and will contain 'size' objects. Each JVM then creates it's own Implementation object, and adds it to the group using the Group.join method. Group.join will block until all 'size' objects have been added to the group. The last JVM (with rank 'size-1') then retrieves a group reference using Group.lookup. This method will return a stub generated by the Ibis compiler. This stub contains the

```
import ibis.gmi.GroupMember;

class Implementation extends GroupMember implements Example {

   private String message = null;

   public synchronized void put(String message) {
      this.message = message;
      notify();
   }

   public synchronized String get() {
      while (message == null) {
         wait();
      }
      return message;
   }
}
```

Figure 8: Implementing a group object with GMI.

communication code necessary to communicate with the objects in the group. Because the stub implements the Example interface, normal invocations of the put and get methods can be used and no communication code needs to be written by the programmer. But before the methods can be invoked, the group stub must first be configured. All methods in a group stub can be configured separately. In this example we will only use the put method. To configure put, we first perform a lookup of the method using Group.findMethod. A GroupMethod object will be returned, which represents the put method of the stub. Using the configure method in this GroupMethod object, it can be specified how the invocations of the put method should be handled. For this purpose the configure method takes two parameters, one describing how the invocation must be forwarded to the group, and one describing how the replies should be returned. In the example application we use GroupInvocation and DiscardReply, which indicates that invocations of put will be forwarded to all objects in the group, and that no replies will be returned. After the configuration is completed, the last JVM invokes the put method. The method invocation is then forwarded to all object in the group. All JVMs then retrieve their local copy of the string by directly invoking the get method on their implementation objects.

7.3 Compiling and running a GMI program

Before running a GMI application it must be compiled. Using *ant*, this is quite easy. Here is an example build.xml file:

```
<project
    name="Project name"
```

```
import ibis.gmi.*;

class BroadcastExample {

   public static void main(String[] args) {
      int size = Group.size();
      int rank = Group.rank();

      if (rank == 0) {
         // JVM 0 creates a new group.
         Group.create("ExampleGroup", Example.class, size);
      }

      // All JVMs create an implementation object.
      Implementation impl = new Implementation();

      // And join the group
      Group.join("ExampleGroup", impl);

      if (rank == size-1) {
         // The last JVM retrieves a group reference
         Example group = (Example) Group.lookup("ExampleGroup");

         // Then configures 'put' to be forwarded to the whole group
         GroupMethod m = Group.findMethod(group, "void put(java.lang.String)");
         m.configure(new GroupInvocation(), new DiscardReply());

         // Now invoke a method on the group
         group.put("Hello world!");
      }

      // All JVMs can now retrieve the data using a local(!) call
      String message = impl.get();
      System.out.println(message);

      // Done
      Group.exit();
   }
}
```

Figure 9: An example GMI application that uses a group object.

```
      default="build"
      basedir=".">

      <description>
      GMI 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-gmi-app.xml"/>
</project>
```

Again, we assume that the environment variable `IBIS_HOME` reflects the location of your Ibis installation.

Invoking *ant build* compiles the application for GMI.

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. To process a GMI application, *ibisc* needs to know that it has to do so. This can be accomplished by passing the `-gmi` flag to *ibisc*.

Running a GMI program is very much like running an Ibis application. See Section 4 for details.

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