Communicating Process Architectures 2000
P.H. Welch and A.W.P. Bakkers (Eds.)
IOS Press, 2000

Distributed Computing using Channel Communications and Java

A Ripke, A R Allen, Y Feng, and S C Allison
Department of Engineering, University of Aberdeen, Aberdeen, AB24 3UE, UK

Abstract. Various methods of connecting CSP channels to external software systems are examined. The aim is to facilitate the implementation of distributed heterogeneous systems using channel communication. The paper concentrates on TCP/IP connections in a Java environment. The approaches used are: custom protocol; object serialization stream protocol; Remote Method Invocation (RMI); Common Object Request Broker Architecture (CORBA). Practical systems are described and compared.

1 Introduction

The rapid development of Web technologies is opening up a number of different possibilities for using the Internet in new ways. One aspect of these changes is the evolution from the information Web to a computing Web. Recent developments in networking and application software are facilitating the use of wide area networks in distributed computing. This will change the way in which some science and engineering is done, allowing researchers to access remotely not only libraries but also instruments and computational resources. With the development of high speed network technologies, high performance computing systems are evolving into heterogeneous networked resources.

In order to permit wide take-up of remote, large-scale parallel computation, there need to be clearly visible benefits and transparent user access. Users will demand *interactive* access to remote parallel applications. The software tools must be put in place to allow researchers to make use of remote systems using commonly available software and widely accepted means of working.

An important class of computational resource consists of systems built using the principles of Communicating Sequential Processes (CSP). Among the advantages of such systems is that they can be relatively easily designed to be free of deadlocks, race hazards, etc. Software implementations include occam, and Java libraries such as CTJ and JCSP. The normal means by which such systems communicate is through *channels*. In this paper, we examine methods of connecting such channels to external software systems. This would facilitate the implementation of distributed heterogeneous systems using CSP channels. The efficient implementation of such a development would, for example, allow a graphical application running on a PC to interact with a parallel computational program running on a remote computer.

2 Background

2.1 Java

Java seems a good choice to use as an application platform for our networking experiments. It offers easy creation of a graphical user interface (GUI), is mostly hardware independent, and is an ideal tool to create interfaces for network applications. In the following we describe several Java based techniques we use with our different experiments. These are Java CSP toolkits, the serialized object stream protocol, and remote method invocation.

2.1.1 Java and CSP

The Java Plug and Play project (JavaPP) [1] introduces the concept of CSP to Java. There are two different libraries available, namely Communicating Threads for Java (CTJ) [2, 3, 4] and Java Communicating Sequential Processes (JCSP) [5, 6]. The current version of JCSP does not provide support for external channels: this feature could be added, but we decided to use CTJ in the present work.

The main subjects of CTJ are external communication and the use of CSP elements in general. CTJ (version 0.9 release 15) offers a variety of external communications like TCP/IP or serial and parallel ports. The channel communication is based on object serialization.

CTJ has introduced new classes for channels like `channel_of_Integer` where `Integer` is not `java.lang.Integer` but re-defined in `csp.lang.Integer` (similarly with all the other primitive types like `Byte`, `Word`, `Long`, etc). But `channel_of_Object`, where `Object` is defined in `java.lang.object`, is available and therefore all objects in Java that implement the serialization interface can be used.

Each channel is associated with a linkdriver that implements the actual communication. This concept is very flexible: new linkdrivers can be easily added to the channel concept of CTJ. We are using the datagram linkdriver for TCP/IP which is publicly available.

2.1.2 Serialized Object Stream Protocol

Serialization is a way to introduce persistent objects to Java that can be stored or transmitted to a Java program running on another platform. "Object serialization in the Java system is the process of creating a serialized representation of objects or a graph of objects. Object values and types are serialized with sufficient information to insure that the equivalent typed object can be recreated. Deserialization is the symmetric process of recreating the object or graph of objects from the serialized representation. Different versions of a class can write and read compatible streams." [7] `java.io.Serializable` is an interface that is implemented by many Java classes. For example, the class `java.awt.Component` is serializable. So complex objects can be stored or transferred in a persistent state.

The protocol is designed to require only stream access to the data. Communication takes place over an `ObjectOutputStream` or `ObjectInputStream`. These two are based on `DataOutputStream` and `DataInputStream` respectively. Only variables that are not defined as transient are transmitted. The programmer can override the methods `readObject` and `writeObject` (in this case the default methods are invoked as `defaultReadObject` and `defaultWriteObject`) to add additional information to the stream. The information being transmitted is not only the variables but also their types.

Another possibility would be to use externalization. With the protocol version 2 that became available in Java 1.1.7, it is possible to skip externalized objects in the stream if the format is unknown on the receiver side. All information is stored in data blocks and the stream is still parseable if the format is unknown.

The serialized object stream protocol is normally only used to connect to other Java applications, or to the filesystem if the object is to be stored. In order to get it working with programs written in languages other than Java, a protocol converter is needed that extracts the information from the channel.

2.1.3 RMI

RMI [8] is the Java equivalent of the remote procedure call (RPC) and is language dependent, as opposed to the language independent RPC mechanism for Unix platforms. RMI is based on the above object serialization stream protocol, and has several useful features like the Java security mechanism. It is a high-level approach and is consequently easy to use.

With RMI and the Java Native Interface (JNI) it is possible to wrap other language modules, like C, into Java procedures. According to the specifications [9], the JNI is a good approach to "legacy" systems. Or in our case it would be a suitable technique to access hardware that is normally not accessible with Java without much effort. But Java is still required on both sides; and additional work is needed to communicate between Java and occam.

2.2 CORBA

In distributed object technology (DOT), a network of software objects collaborate to provide computational services. Multiple objects can be dynamically created as required for the solution of a particular task, and these objects can reside on heterogeneous architectures, on a local network or across the Internet, and can be programmed in a variety of languages. CORBA, the Common Object Request Broker Architecture [10], is a distributed object standard developed by the Object Management Group (OMG) and their corporate members and sponsors. OMG standards are endorsed by the ISO, and by X/Open as part of the Common Application Environment specification.

The CORBA model of distributed computing is essentially peer-to-peer. Individual components can act as clients or servers, in the sense that a client makes a request from another component, and a server provides an implementation of an object required by a client. The CORBA specification describes a kind of software bus, called an *Object Request Broker* (ORB), which provides an infrastructure for distributed object computing. It enables client applications to communicate with remote objects, invoking operations either statically or dynamically. The ORB deals with the identification and location of objects, handling of connection management, and delivery of data.

The OMG Object Model defines common object semantics for specifying the externally visible characteristics of objects in a standard way. In this model, clients request services from objects (servers) through a well-defined interface. A client accesses an object by issuing a request to the object. The request is an event, which carries information including an operation (or method), the object reference of the service provider, and actual parameters (if any). Distributed objects in a CORBA application are described in *Interface Definition Language* (IDL), which allows transparency in terms of both location and programming language. The IDL provides a platform- and implementation-independent way to define what operations an object supports by specifying their interfaces. An interface consists of a set of named operations and the parameters to these operations.

An object implementation uses an *object adapter* to make itself available through an ORB, and which the ORB uses to manage the run-time environment of the implementation. To ensure that different ORBs can communicate effectively, an *interoperability architecture* is defined. This includes a common object reference format and protocol interoperability definitions. In particular, the Internet inter-ORB protocol (IIOP) specifies how ORBs can communicate using TCP/IP connections.

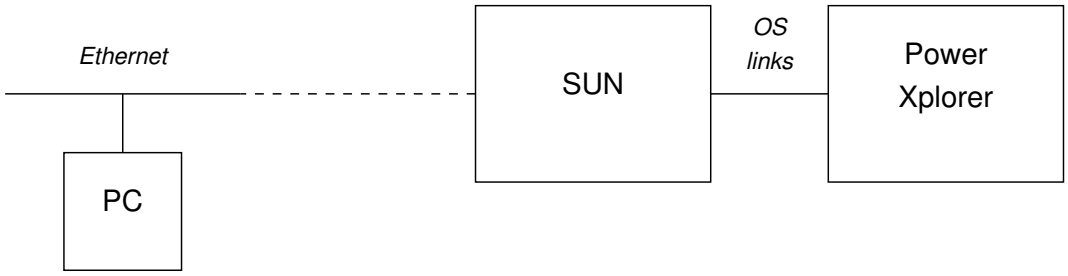

Figure 1: Hardware setup.

There are a number of utility services which applications often require in a distributed framework, collectively known as *CORBAservices*. These cover such areas as Naming, Trading, Transactions, Security, Time and so on. For example, the Naming Service provides human-readable object identifiers and persistent object identification.

3 Practical investigation

3.1 Experimental setup

The parallel computer used is a Parsytec PowerXplorer with 16 PowerPC601 nodes with 8 MB RAM each. We have ported [11] the Kent Retargettable occam Compiler (KRoC) [12, 13] to the Parsytec box. This port has an extension which allows channels to connect to the Internet via TCP/IP sockets. The parallel computer is connected to a Sun workstation via an S-bus board providing transputer links. The host provides hardware resources like keyboard, screen, filesystem and Internet connection (Figure 1). occam programs are run on the Xplorer (using KRoC) and on the Sun (using SPOC [14, 15]).

Inprise VisiBroker for Java (Solaris version) is used to provide the CORBA environment and to develop a variety of interface programs, object server programs and applet applications on the front-end host.

A PC runs a Web browser to invoke remote occam applications and display the results of the application via applets. In order to run CORBA-compliant Java applications on a PC Web browser, Inprise VisiBroker for Java (Win95 version) is used in the PC to provide the runtime support. The Sun and PC are connected via a 10Mb/s ethernet network.

3.2 Approaches

The following four approaches are examined in this section.

- Custom protocol
- Object serialization stream protocol
- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA).

Only TCP/IP connections will be considered because it seems to be the most suitable and flexible form for remote computing, at least in our case. Apart from discussing the form of the protocol, it is also important to review additional aspects of the communication: for example, the necessity or otherwise of having an additional application running on the host computer providing the external communication. This means in our case whether it is worth having an extra program running on the Sun handling the external communication, or dealing directly with the PowerXplorer array.

3.3 Custom Protocol

A custom protocol can be a simple TCP/IP connection with sockets on both the occam and Java sides. This approach is language- and system-independent in the sense that it uses standard Unix sockets. It is a quick solution and provides fast and flexible data handling with a minimal overhead. The drawback is that the programmer has to design his own protocol that is not compatible with any standards. Both sides, occam and Java, have to know about the protocol. The endian problem has to be solved, too, if the underlying architecture uses both little and big endian. Java uses big endian; KRoC on the PowerXplorer is designed for little endian, following the occam specification. (Although the PowerXplorer architecture supports both endians.)

A connection between the custom protocol and CTJ can be built with an appropriate TCP/IP linkdriver, written for a user-defined protocol that wraps up the object serialization stream protocol used by CTJ. The linkdriver itself simply uses plain sockets. The programmer is fully responsible for the format and design of the protocol.

The following listings give a very simple example of communication between a multi-processor occam application and Java using bare sockets. Below are the KRoC network configuration file and two occam processes running on the Xplorer, using channels mapped onto sockets. Process `proc1` reads in 30 bytes from the computer *galileo* and forwards them to the screen and via an internal channel to process `proc2`. The second process runs on another processor and sends the received bytes back to *galileo*.

```
#USE "proc1.tco"
#USE "proc2.tco"

CLIENT STREAM sokin  READS  "galileo:20001":
CLIENT STREAM sokout WRITES "galileo:20002":

CHAN OF BYTE c:

CONFIG
  PLACED PAR
    PROCESSOR one
      proc1(stdout,sokin,c)
    PROCESSOR two
      proc2(sokout,c)
:
```

```
PROC proc1 (
  CHAN OF BYTE stdout,
  CHAN OF BYTE sokin,
  CHAN OF BYTE c
)
  BYTE a:
  SEQ
    SEQ i = 1 FOR 30
      SEQ
        sokin ? a
        c ! a
        stdout ! a
    stdout ! 255
:
```

```
PROC proc2 (
  CHAN OF BYTE sokout,
  CHAN OF BYTE c
)
  BYTE a:
  SEQ
    SEQ i = 1 FOR 30
      SEQ
        c ? a
        sokout ! a
:
```

Finally, the Java application running on *galileo*:

```
import java.net.*;
import java.util.*;
import java.io.*;

public class occamconnect{
  public static void main(String args[]){
    try {
      ServerSocket servsokout = new ServerSocket(20001);
      ServerSocket servsokin = new ServerSocket(20002);
      Socket sokin = servsokin.accept();
      Socket sokout = servsokout.accept();
      DataOutputStream sout =
                          new DataOutputStream(sokout.getOutputStream());
      DataInputStream sin = new DataInputStream(sokin.getInputStream());
      sout.writeBytes("01234567890123456789012345678 9");
      sout.flush();
      sout.close();
      sin.readFully(new byte[30]);
      sin.close();
    }
    catch (UnknownHostException f){System.out.println(f.toString());}
    catch (IOException e) {System.out.println(e.toString());}
  }
}
```

3.4 Object Serialization Stream Protocol

In order to use this protocol between Java and occam we considered two different approaches. One is manual—the other semi-automatic.

The first approach is simply typing the appropriate stream control tokens into the source code for the object that is supposed to be transferred. With the second approach, an object is sent from a Java program to a protocol interface generator that parses the stream and writes all the control tokens into a piece of occam source code. This file has to be included into the actual occam program.

The fact that the protocol is designed to require only stream access to the data is important for the protocol interface generator, as there is no need to store objects from the stream temporarily. Each distinct object has got its own number (`serialVersionID`). This number can be gathered either by scanning a stream or by using the Java program *serialver*. Because the class field names are explicitly transferred in the class descriptor of the stream, the occam channel protocol can be generated with meaningful names.

This approach cannot be fully automated because of the static nature of occam: it is not possible to create a dynamic channel protocol at run-time. Therefore, the protocol translator has to be re-created whenever the types of objects have changed. Of course, there are other possible approaches using this protocol. Instead of using an external interface generator or changing it manually, a protocol converter can be either written in occam or integrated into this specific KRoC port as a new channel type.

The protocol interface generator consists of two parts: a system- and language-independent frontend that parses the stream protocol, and a system- and language-dependent backend for automated code generation. This interface creator is not only focussed on translating from Java to occam: it could be adapted to produce C code.

Different versions of this approach could be the installation of Java on the PowerXplorer, or the use of Java on the host machine which receives the object over a CTJ channel and

forwards the relevant information to the KRoC program. But here again we need a protocol translation that requires a new kind of protocol to communicate between Java and occam. So the solution will again be a custom protocol that has to be changed every time the objects change. This different version would only introduce an additional overhead to the communication and should not be considered any further.

This approach can be used easily with the CTJ library and introduces only little overhead. On the Java side there is no additional effort involved. One drawback is the conversion from big endian to little endian and vice versa. This can be done in the protocol translator interface and should not always be required.

The following code example shows how a 256×256 byte array is sent over an external channel using the serialized object stream protocol. Because it is a two dimensional array the transfer happens row-wise. Firstly, the whole two dimensional byte array is initialised in the stream as an array with 256 elements which are the rows. Secondly, the first row is set up as an array with 256 elements which are the bytes in the row; the first row is sent. Thirdly, each of the remaining rows is initialised by referring to the previous array description before transfer.

```
PROC transfer (CHAN OF ANY sokout, CHAN OF BYTE stdout, CHAN OF ANY in1)
  SEQ
    ... PROC print
    ... variables
    ... constants
    SEQ
      --{{{  send an array (256x256)
      --{{{  constants for serialisation
      VAL START IS [#AC,#ED,#00,#05]:     -- magic and version
      VAL ARRAY.BYTE.BYTE.NAME IS [#00,#03,#5B,#5B,#42]:
      VAL ARRAY.BYTE.NAME IS [#00,#02,#5B,#42]:
      VAL SERIAL.VERSION.UID.2 IS [#AC,#F3,#17,#F8,#06,#08,#54,#E0]:
      VAL SERIAL.VERSION.UID.1 IS [#4B,#FD,#19,#15,#67,#67,#DB,#37]:
      VAL CLASS.DESC.FLAGS IS BYTE #02:
      VAL FIELDS IS [#00,#00]:
      VAL ARRAY.SIZE IS [#00,#00,#00,#FF]:
      VAL TC.CLASSDESC IS BYTE #75:
      VAL TC.ARRAY IS BYTE #74:
      VAL TC.NULL IS BYTE #70:
      VAL TC.REFERENCE IS BYTE #71:
      VAL END.BLOCKDATA IS BYTE #78:
      VAL SECOND.OBJECT.REFERENCE IS [#00,#7E,#00,#02]:
      --}}}
      VAL width IS 256:
      VAL height IS 256:
      [width][height]BYTE buffer:
      SEQ
        in1 ? buffer
        --{{{  initialisation whole array
        sokout ! START
        sokout ! TC.ARRAY
        sokout ! TC.CLASSDESC
        sokout ! ARRAY.BYTE.BYTE.NAME
        sokout ! SERIAL.VERSION.UID.1
        sokout ! CLASS.DESC.FLAGS
        sokout ! FIELDS
        sokout ! END.BLOCKDATA
        sokout ! TC.NULL
        sokout ! ARRAY.SIZE
        --}}}
```

```
    --{{{  initialisation first row
    sokout ! TC.ARRAY
    sokout ! TC.CLASSDESC
    sokout ! ARRAY.BYTE.NAME
    sokout ! SERIAL.VERSION.UID.2
    sokout ! CLASS.DESC.FLAGS
    sokout ! FIELDS
    sokout ! END.BLOCKDATA
    sokout ! TC.NULL
    sokout ! ARRAY.SIZE
    --}}}
    --{{{  sending first row
    sokout ! buffer[0]
    --}}}
    SEQ i=1 FOR height
      SEQ
        --{{{  initialisation other row
        sokout ! TC.ARRAY
        sokout ! TC.REFERENCE
        sokout ! SECOND.OBJECT.REFERENCE
        sokout ! ARRAY.SIZE
        --}}}
        --{{{  sending row
        sokout ! buffer[i]
        --}}}
--}}}
```

3.5 Remote Method Invocation (RMI)

We have used RMI to link a GUI to a remote parallel application [16], see Figure 2. A Java program runs on the Sun to act as an RMI server. A graphical user interface running on the user's machine allows the user to choose a configuration for the Xplorer parallel system. If a particular hardware configuration is not available, the user can use a graphical tool to select a new one. The network configuration file for KRoC is automatically generated, and transferred to the Sun where it can re-compile the occam application. The remote configuration and startup of the occam program is achieved with RMI and `java.lang.Runtime.exec()`. Once the occam program has started, sockets are started to handle data transfer between occam and the user's machine using a custom protocol.

The endian problem does not have to be solved during communication between both Java sides, but between Java and occam. A new implementation without using Java on one side seems to be quite a complicated task. RMI adds some further information to its protocol compared to the object serialization stream protocol.

3.6 CORBA

Parallel applications are programmed in occam and run on the Sun host (using SPOC) or the PowerXplorer multiprocessor system (using KRoC). As previously, sockets are used to connect occam channels to the outside world. The Sun hosts the DOT environment, a Web server, and two interface programs (dealing with invocation of parallel program and handling of results respectively). A PC connected to the network runs a Web browser. Applets running on the browser ask for remote computing services and display results.

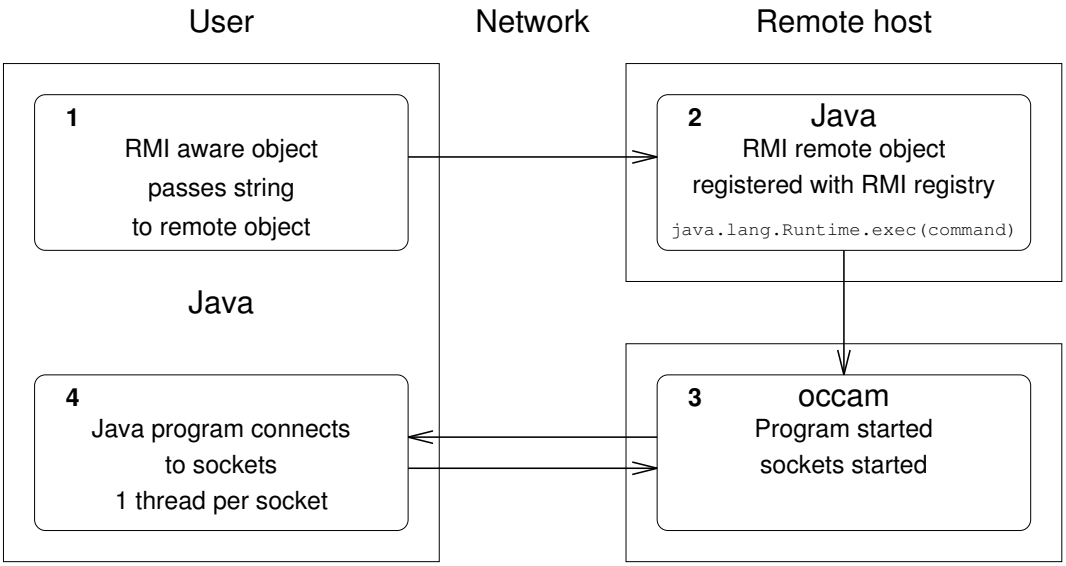

Figure 2: RMI interface.

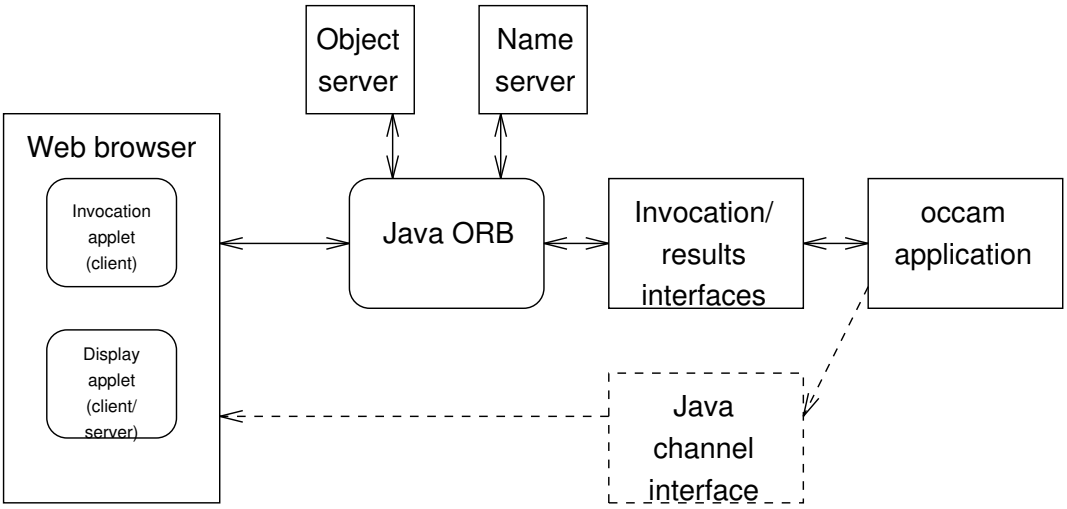

Figure 3: Simplified software structure using CORBA.

As previously mentioned, Oc-X, our port of KRoC for the PowerXplorer, can be configured so that channels are mapped onto sockets. In addition, SPOC running on the Sun was modified to allow an occam channel to connect to a socket. This was done by modifying the libhostio.a library. In particular, `so.write.string()` and `so.write.int()` were extended by adding a new parameter `VAL [] BYTE portend`, to allow strings and integers to be written to a socket.

The basic software structure is shown in Figure 3. A Java program provides the interface between the parallel application and the DOT environment. Remote invocation of the application is implemented with a CORBA object. In order to experiment with a variety of software architectures, the invocation and results display were implemented as separate applets. In all cases, the invocation applet acts as a CORBA client, sending a request which is served by the invocation object, which starts the occam application. Three experimental system structures were investigated [17]:

A. Both the invocation and display applets act as CORBA clients. The display client sends a request which is served by the results object, which passes the output of the occam application to the display applet as the return value.

B. The display applet in this case acts as an object providing a display service. The results interface acts as a CORBA client which requests display through the CORBA Naming Service.

C. The display is done by directly connecting the occam application output to a display applet via a CTJ channel.

3.6.1 Remote invocation

A CORBA object is implemented for obtaining remote parallel computing services. This object provides operations such as remote invocation of parallel applications and other management and control operations. The following is part of the IDL file of the object definition.

```
module Execute {
  interface Running {
    void occamexec(in string application);
    string OAname();
    void Namewrapper1();
    void Namewrapper2();
    ...
  };
};
```

The `Running` interface includes the following operations: `occamexec(in string application)` executes the occam application specified by a CORBA parameter, `application`; `OAname()` provides some information about the occam applications on the parallel computer; `Namewrapper1()` etc, invoke the output wrappers of occam applications.

The IDL compiler, `idl2java`, directly compiles the IDL files into the implementations of stubs and skeletons in Java. Each interface will create its own stub and skeleton classes. As Java allows only one public interface or class per file and there are some differences between the Java language specification and CORBA specification, compiling `Execute.idl` generates several Java files. One of these is `Running.java`: this declares a Java interface

```
public interface Running extends org.omg.CORBA.object
```

which includes methods corresponding to the CORBA operations:

```
public void occamexec (java.lang.String application)
public java.lang.String OAname()
public void Namewrapper1()
public void Namewrapper2()
...
```

The client stubs are instantiated as local proxy objects that delegate invocations on their methods to the remote implementation, through two Java classes, `org.omg.CORBA.portable.OutputStream` and `.InputStream`. Using the methods of these two classes, such as `_invoke()`, `_request()`, `read_<type>()`, `write_<type>()`, the invocation of the methods of remote objects, the passing of parameters, and the return of results, are implemented.

Remote invocation is implemented through the methods of a CORBA object residing on the front-end host of the parallel computer. To instantiate the object implementation and to make it available to clients, an object server is implemented. The object server has four fundamental tasks:

- Initialising the environment, that is, getting the reference to the pseudo-objects for the ORB and BOA (Basic Object Adapter)

- Creating objects
- Making objects accessible to the outside world
- Executing a dispatch loop to wait for invocations.

The server for the `Running` object includes the following code:

```
public class Executeserver {

 public static void main(String[] args) {
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    org.omg.CORBA.BOA boa = orb.BOA_init();
    Execute.Running running = new RunningImpl("Occamrunning");
    boa.obj_is_ready(running);
    boa.impl_is_ready();
    ...
  }
}
```

The implementation of the `Running` object extends the skeleton generated by idl2java. This class provides the concrete implementations of the methods. For example:

```
public void occamexec(String application) {
  Runtime rt = Runtime.getRuntime();
  try {
       Process P1 = rt.exec(application);
  } catch (IOException e) {
    System.err.println("exec error:" +e);
    }
}
```

This wrapper uses the Java `Runtime` class's `exec()` method to convert the invocation of an occam application into a method call of a CORBA object. The command line entered by the user in the PC browser is passed to the front-end host as a string parameter `application`.

The client application is designed to be a Java applet that can be run from any Java-enabled browser. The applet sets up the GUI environment, and initialises the Object Request Broker. It then invokes methods of the `Running` object in response to user mouse clicks and text input.

3.6.2 Results display

In structure A, the display applet is a CORBA client. The wrapper which interfaces between the output of the occam application and the CORBA ORB acts as an object implementation. The display applet sends a request for the service of the wrapper object. The object implements the method of the service and returns the result. That is, the applet gets the result returning from the service object only after sending its request and the object finishing its service method. This structure has been implemented, but the display applet can obtain only static output results from occam parallel applications.

In order to achieve dynamic display, in structure B the display applet acts as a server providing display services, which receives display requests from the client (an interface wrapper). The results data are obtained as the parameter of the client request.

As part of the purpose of this work was to investigate the system's behaviour within a general CORBA context, the display server was set up using the CORBA Naming Service. This

permits, for example, the adoption of hierarchical naming schemes. A Java class `NamingMethods` was implemented which enables CORBA objects and clients to get an object reference to a `NamingContext` and then use string arguments with the "/" character as a name separator to identify objects relative to that context, such as `/Par/Occ/Myclient/Myobject`.

Our final method of results display achieves this by directly connecting the occam application output to a display applet via a CTJ channel, using the TCP/IP link driver.

3.6.3 Interface programs

The interface wrappers are key facilities which interface the outputs of occam applications with the DOT environment. Through these interface programs, occam parallel applications can use the services provided by distributed objects on the Web to display results.

As the outputs of occam applications use different sockets and there are different data types that interface programs pass, several interface programs were developed. We have developed interface wrappers which support (a) stream sockets (for KRoC), (b) datagram sockets (for SPOC).

3.6.4 Test applications

A web browser running on a PC was used to run the invocation and display applets (Figure 4). These were used to start up occam applications on a remote host and display results. All the experimental structures worked correctly. The test applications used were: (i) `CItest`: occam sending character strings and integers to check the behaviour of the modified SPOC; (ii) `comstime`: standard communications test program; (iii) `parhello`: the outputs of multiple occam processes displayed by multiple applets; (iv) `clock`: a graphical analogue clock display is updated by a remote occam program.

Structure A provides only limited display capabilities: it is difficult to implement a dynamic display. Structure B, using a display server applet, is the natural way to achieve this.

The clock display was implemented using both structure B and structure C. Structure C shows that CTJ channels can be integrated successfully into a DOT environment: however, the target of the link driver must be defined explicitly in the program, thus losing the location transparency of CORBA.

4 Discussion

Experiments with a custom protocol have shown that the transfer (over 10Mb/s ethernet) of a 1MB array of `Byte` between a remote PC and the PowerXplorer running a KRoC program takes 3.0 seconds if using C and plain sockets, 3.7 seconds with Java and `java.io.DataInputStream.readFully`. When using the serialized object stream protocol, the overhead for sending a 1MB array of `Byte` is only 46 bytes and nearly the same results as for the custom protocol are achieved. CORBA communications are built on top of sockets, and therefore there is naturally a price to pay in terms of communications speed. From our tests, it would appear that CORBA imposes a performance overhead of between a factor of 2 and 5, compared with a direct CTJ channel.

The DOT approach provides the benefits of location transparency of objects. By defining interfaces independently of implementations, it abstracts away from architecture and language details. It supports the concept of a network of heterogeneous implementations being used as a distributed computational resource. CORBA also provides automatic code generation to deal with remote invocations; and access to standard CORBA services and facilities.

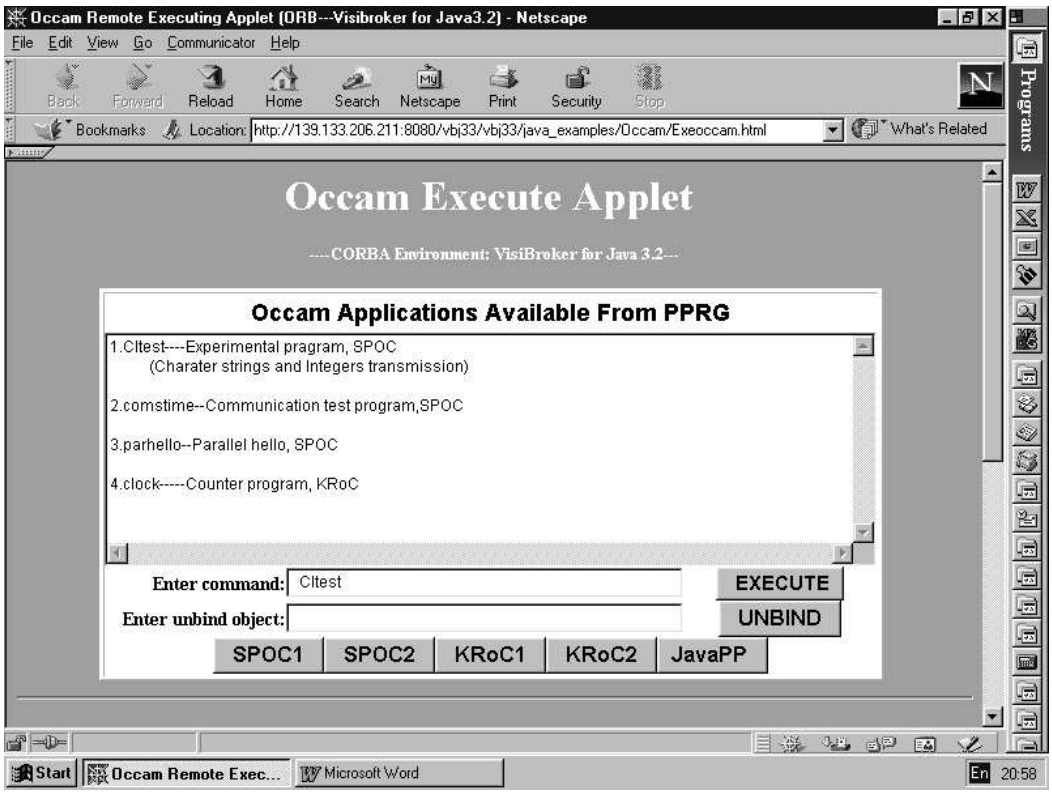

Figure 4: Applet for CORBA remote invocation.

Using the techniques presented here, it is feasible to connect to CSP channels from within such an environment. There is an initial learning curve with CORBA, but once the principles are understood, it is possible to get simple distributed applications going fairly readily. The programmer is helped by the software tools which create many of the necessary files automatically. We have not attempted to optimise the implementation in any way, and undoubtedly it could be made more efficient. Nevertheless, it is clear that DOT does impose overheads, and at present it could only be recommended in cases where there is a high computation/communication ratio: for example, in interacting with large parallel applications where there may be a large degree of internal concurrency but with relatively small I/O requirements.

From the above protocols only CORBA and the custom protocol provide a neutral communication protocol. But CORBA needs an interface definition language for the underlying language. The custom approach requires a new protocol design for each application. RMI and the serialized object stream protocol are language dependent. But at least for the latter case an interface for a different language can be created without too much effort. Only the custom protocol does not use a standard and is therefore the most inflexible approach when it comes to porting the model to other hardware/software configurations. The serialized object stream protocol seems to offer the best solution by producing a direct connection without requiring an extra interface program.

Even though the serialized object stream protocol seems to be quite complicated to use manually, once there is an automatic interface generator available it should be powerful and easy to use a channel protocol to connect occam programs to Java interfaces.

References

[1] *JavaPP*. http://www.cs.bris.ac.uk/~alan/javapp.html.

[2] G. Hilderink et al. Communicating Java Threads. In A. Bakkers, editor, *Parallel Programming and Java*, pages 48–76, Amsterdam, 1997. IOS Press. ISBN 90-5199-336-6.

[3] G. H. Hilderink. Communicating Java Threads Reference Manual. In A. Bakkers, editor, *Parallel Programming and Java*, pages 283–325, Amsterdam, 1997. IOS Press. ISBN 90-5199-336-6.

[4] G. H. Hilderink. *Communicating Threads for Java (CTJ)*. http://www.rt.el.utwente.nl/javapp.

[5] P. H. Welch. Java threads in the light of occam/CSP. In P. H. Welch and A. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 259–284, Amsterdam, 1998. IOS Press. ISBN 90-5199-391-9.

[6] P. H. Welch. *CSP for Java*. http://www.cs.ukc.ac.uk/projects/ofa/jcsp.

[7] *Java Object Serialization Specification*, 1998. ftp://ftp.javasoft.com/docs/jdk1.2/serial-spec-JDK1.2.pdf.

[8] *Java Remote Method Invocation Specification*, 1998. ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf.

[9] *Java Native Interface Specification*, 1997. ftp://ftp.javasoft.com/docs/jdk1.1/jni.pdf.

[10] Object Management Group. *CORBA*. http://www.corba.org.

[11] T. M. Sheen, A. R. Allen, A. Ripke, and S. Woo. oc-X: an optimising multiprocessor occam system for the PowerPC. In P. H. Welch and A. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 167–186, Amsterdam, 1998. IOS Press. ISBN 90-5199-391-9.

[12] D. C. Wood and P. H. Welch. The Kent retargettable occam compiler. In B. C. O'Neill, editor, *Parallel Processing Developments*, pages 143–166, Amsterdam, 1996. IOS Press. ISBN 90-5199-261-0.

[13] *KRoC*. http://wotug.ukc.ac.uk/parallel/occam/projects/occam-for-all/kroc.

[14] M. Debbage, M. Hill, S. Wykes, and D. Nicole. Southampton's portable occam compiler (SPOC). In R. Miles and A. Chalmers, editors, *Progress in Transputer and occam Research*, pages 40–55, Amsterdam, 1994. IOS Press.

[15] *SPOC*. http://gales.ecs.soton.ac.uk/software/spoc.

[16] S. C. Allison. Java and concurrency. Master's thesis, University of Aberdeen, 1998.

[17] Yibing Feng. Integration of parallel computing and the web environment with distributed object technology: an experimental development. Master's thesis, University of Aberdeen, 1999.