Application of Java and CORBA to Distributed Control and Monitoring Applications in the PHENIX Online Control System

E. Desmond[1], *S. Adler*[1], *Lars Ewell*[1], *J. Haggerty*[1], *Hyon Joo Kehayias*[1], *S. Pate*[2], *Martin L. Purschke*[1], *R. Roth*[1], *C. Witzig*[1],

[1] Physics Department, Brookhaven National Lab,USA
[2] New Mexico State University, Las Cruces NM,USA

Abstract

The integration of Java with CORBA to achieve platform independent distributed control and system monitoring applications in the PHENIX Online Computer System will be presented. The PHENIX Online Computer system is distributed over the Solaris, NT, Linux, and VxWorks operating systems. These environments run on Sun SPARCstation, Pentium and Power PC architectures respectively. CORBA has been used extensively in this environment to provide distributed access among these divergent platforms.

This paper will present the experiences gained and the techniques, which were developed in applying these two technologies to the PHENIX, distributed control environment. Techniques used for passing complex data types between clients and server applications will be presented. Performance and Security issues of Java based applications and applets and their relevance to the control environment of large-scale detector requirements will be discussed. Java provides the promise of platform independence by executing bytecode in a uniform Virtual Machine. The OMG CORBA IIOP communication protocol was developed to provide seamless communication of applications developed with different ORB implementations. The experience and degree to which these promises of ORB interoperability, platform independence and seamless communication in the PHENIX control and monitoring applications will be evaluated

Keywords: Java,CORBA,control system

1 Introduction

Java is a highly productive development environment for building robust, extensible enterprise wide applications. The rich set of class libraries for display, and its clear object oriented architecture design make Java a highly desirable framework for the development of online control and monitoring applications. CORBA is a framework for the development of platform independent distributed computing applications. It provides a set of components and protocols that allow the development of platform independent and language independent access to remotely distributed components. By combining these technologies together, enterprises can potentially develop portable, platform independent distributed applications

2 Use of Java in the PHENIX On-line System

In the PHENIX online system, Java is being used extensively for the development and implementation of the major detector control and monitoring applications. These applications include the run control application which is the main user interface for configuring, controlling and running the PHENIX detector. Other user interface applications include programs for diagnostics and control of various hardware components, including the embedded data collection modules (DCMs), the front end signal processing modules, the event builder, level one electronics. Other applications currently in development including those for system partitioning and error logging and display, will similarly use this technology.

While Java is the language of choice for client side user interface development, CORBA is the standard mechanism for communication among distributed components in the PHENIX online system[1]. The server and object implementation code is written in C++. Servers have been implemented on sparc, Pentium and Power PC platforms which are running the Solaris, NT and VxWorks operating systems, respectively. In addition, the PHENIX counting house has a number of Linux based Pentium system from which operators will communicate with the control system. All server application objects were compiled from IDL files with IONA Technologies Orbix2.3c compiler. IONA's OrbixWeb version 3.2 is being used for IDL compilation to Java code[2]. The goal was to develop Java based application which could execute on these different platforms and would transparently communicate with all PHENIX servers. This paper describes the extent to which that has bee achieved

3 Portability

With the introduction by the Object Management Group (OMG) of the CORBA 2.0 specification, the Internet Inter-ORB Protocol (IIOP) became part of the CORBA specification. IIOP is a special case of the General Inter-orb Protocol (GIOP) which is implemented on top of TCP/IP. In addition to IIOP, the OMG also specified a standard object format which is also required for ORB interoperability. This object format is called the Interoperable Object Reference (IOR). The IOR is a representation of an object which contains all the information which is necessary for a application to connect to an object and make a remote invocation on the object's methods. The IOR encodes the type of object, the object host and port number of the server for the object, and an object key which is used by the server to locate the object. By using the IOR over IIOP, objects can communicate with objects which were compiled from a IDL compiler from different ORB vendors. In addition, with these protocols it is not necessary to run a vendor's ORB daemon on the client host[3].

In order for a Java client application to gain access to a CORBA compliant object, the IOR must be made available to make an initial connection into a CORBA system. Once a client has a reference into a server, this server can make available other object references. In the PHENIX online system, this server is a Naming Service. Objects in this system register themselves with the Naming Service. Client applications then obtain the object reference for a given object name. The reference to this server is made available to Java clients by having the Naming Service create a stringified object reference of itself. This is an object reference which is converted to an ASCII string which can then be written to a file, database or made available by some other means. Once this is done, Java based client applications read the file, reconvert the string to an object reference, and invoke the desired method on the object.

3.1 InterORB Connectivity

The example code below in Listing 1 shows how Java client applications, which are compiled from ORB vendors other than IONA, reference objects from the PHENIX online system. In this application, the IOR is first read from a file and is then converted to an object reference with

[1]Desmond, Ed."PHENIX On-Line Distributed Computing System Architecture", CHEP 97, Berlin, Spring 1997.

[2]Orbix 2.3c; IONA Technologies LTD; The IONA Building 8-10 Pembroke Street, Dublin 2, Ireland `http://www.iona.com`

[3]Michi Henning and Steve Vinoski. 1999. "Advanced CORBA Programming in C++ ", Reading, MA: Addison-Welsey

the string_to_object(String ior) method. Once this is operation is complete, the reference must be narrowed to the appropriate object reference type. Finally, the desired method on the object is invoked. This mechanism has proven itself to effectively provide access to PHENIX object references from Java applications which are compiled with both the Visibroker IDL compiler[4], and the Omnibroker IDL compiler. Java applications which are generated using the Visibroker IDL compiler on NT, and the Omnibroker IDL compiler on Linux have successfully been able to gain access to IIOP compatible servers in the PHENIX online system.

There are a number of limitations to using this method to interORB connectivity that must be kept in mind. First, the client application must know the desired object type to which the reference must be narrowed, and ,for static binding, that the client application have knowledge of the object type at compile time. This requires that the client have access to the IDL file in which the object type is defined. Moreover, when an object reference is generated it contains information about the server from which the object is accessible. This server information includes the port number which the server is listening for CORBA connections. If the server was stopped and restarted since the object reference was written to the file, the port number on which the server is listening for connections may have changed. In addition, at present, the version of Orbix which is currently being run on processors which are running the VxWorks operating system, does not yet support IIOP. While these servers are accessible to Java applications which are run on all the above platforms, the Java generated code must at present be compiled with IONA's OrbixWeb IDL compiler.

Listing 1.

// initialize the orb omg.org.CORBA.ORB orbRef = ORB.init(”phoncs0.phenix”,null); Object objRef; nameserv nsRef;

// open the file ”nsref.dat” in the directory CUTILITIES/src // convert the string which was read to an object reference // and then invoke the method getId(long id) on the reference

// open the text file

byte buff[] = new byte[1000];

try InputStream fileIn = new FileInputStream(”nsref.dat”); int i = fileIn.read(buff);

catch (FileNotFoundException e) System.out.println(”Could not open file nsref.dat ”); catch (IOException e) System.out.println(”IOException ” + e.toString());

// first convert the byte array to a string object String iiopStrRef = new String(buff);

// convert the string to a reference objRef = orbRef.string_to_object(iiopStrRef);

// now narrow the reference to a nameserv object nsRef = nameservHelper.narrow(objRef);

System.out.println(”ACCESS OBJECT NameServ”); // finally invoke a method on the object

int myid = 0;

org.omg.CORBA.IntHolder intHolder = new IntHolder(); myiiopRef.getId(intHolder);

myid = intHolder.value;

System.out.println(”Id of nameserv = ” + myid);

4 Performance

4.1 Time to initialize connection

When considering the use of Java for monitoring and control applications, performance becomes a relevant issue. Accordingly a series of performance measurements were made. These measurements compare the relative performance of equivalent functions in client applications, which were

[4]Visibroker; Inprise; 100 enterprise Way, Scotts Valley, CA (`http://www.inprise.com`

written in Java and C++. The first measurements are the time it took to establish a connection between Java and C++ client applications and CORBA servers[5]. In the first case both clients were running on the same Solaris platform as the server. In the second case, both client applications were run from a remote NT platform. The results of these measurements is shown below in Table 1.

Table I: Connect time from client to C++ Server

Client Platform	Java Client	C++ Client
Solaris (local client)	406 ms	139 ms
NT	915 ms	60 ms

As can be seen, the Java client applications take considerably longer to establish a remote connection. This time does not measure the time it takes for a Java GUI client to paint its display on a console terminal. This paint operation, in general, takes approximately 6 seconds, depending on the complexity of the display. While the connect times are considerably slower for Java than for C++, in general, these connections are made only once when a client first connects to a server. Thus they are not prohibitive for use in a control environment.

4.2 Data Throughput

A second set of measurements were made to measure data throughput between Java clients and CORBA servers5. In these tests different data types were first sent one way from client to server and then the data was both sent and returned from the server to the client. The data types varied from Java native primitive data types to sequences of these types, to structures and finally sequences of structures. In one case the client was written in Java, while in the second case the client was written in C++. In both cases the server was the identical server which was written in C++. In all cases 100 trials were made for each test and the performance was averaged over those trials. The server always ran on the same Solaris Enterprise 4000 sparcStation. Table 2 below shows the averaged elapsed time in milliseconds and throughput in bytes per second.

Table II: Margin specifications

Data type	Java client	C++ client
One way no params	2.6 ms	.7 ms
One way primitive	1.4 ms 17857 bytes/sec	.7 ms 35714 bytes/sec
Oneway structure	1.31 ms 19083 b/s	.7 ms 35714 b/s
Oneway primitive sequence	4.6 ms 543478 b/s	3.71ms 673854 b/s
Oneway struct sequence	4.91 ms 509164 b/s	4.31 ms 580046 b/s
Oneway struct array		4.71 ms 21231 b/s
Twoway primitive	6.21 ms 16103 b/s	4.71 ms 21231 b/s
Twoway structure	4.21 ms 23752 b/s	4.31 ms 23201 b/s
Twoway primitive sequence	23.33 ms 428632 b/s	20.93 ms 477783 b/s
Twoway structure sequence	28.04 ms 356633 b/s	24.91 ms 400962 b/s
Twoway structure array	29.44 ms 339673 b/s	25.54 ms 391542 b/s

NOTE: * indicates that a sequence is equivalent to an array in Java and so this measurement is redundant.

[5]IONA Technologies LTD; The IONA Building 8-10 Pembroke Street, Dublin 2, Ireland `http://www.iona.com`

Generally the performance in transferring data between C++ clients and servers is faster than in Java. However, for control and system monitoring applications, Java has proven to provide completely adequate performance.

5 Security

At the present time, all Java based client programs are being developed as applications only. There are two reasons for not developing Java client applications which may run as applets. First is that when an applet runs in a browser, it is subject to all the restrictions that the security manager applies to ordinary applets. That is, the applet may only access those resources from the server on which it originated. Thus to access other remote servers, via CORBA, the operation would have to request permission from the security manager. Each applet must, in addition have an authentication certificate which is verified by the server. These restrictions introduce additional complexity to Java application development. Secondly, there have been a number of recent security breaches at Brookhaven National Lab as well as at other National Labs. These intrusions include incidents of hacker infiltration of workstations and the introductions of viruses. Accordingly it has become policy that there be no control of the PHENIX facility from remote locations. As only monitoring operations would be permitted from remote sites, the additional effort in supporting the ability to run applets, while at the same time restricting their access to monitor only functions would require development resources and effort which we do not presently have available.

6 Passing complex data types from Java clients

In Java, all parameters that are passed to methods are passed by value. Modified parameter values, which are returned through a function argument, must be contained in holder objects. When this is done the holder object reference is passed by value while the contents of the object may be modified. The same rule applies to remote operations that take place via CORBA calls. Thus all return parameter values, from remote CORBA objects, must be included in a wrapper function.

For each user defined interface and type definition which is included in an IDL file, the IDL compiler will generate three classes, an interface class, a holder class and a helper class. The interface class provides to the client the interface to the proxy class of the remote object. The helper class provides helper functions to allow binding to remote objects and to insert and extract user defined types from CORBA type Any data types. The holder classes provide the wrapper in which to return modified values to remote clients. The following example shows how an IDL file defines an unbounded sequence of strings which is then retrieved from a remote object.

The IDL file contains the following definition definition:

```
typedef sequence (String) seqString ;
interface ns { void op( out any data);
};
```

This results in the creation of a Java class called a seqStringHolder. To return a sequence of strings from a remote member function the following code sequence is required: The following code snippet show how this code is used in a Java application;

```
SeqStringHolder stringholder = new seqStringHolder();
// get reference to nameserver
nsRef.op(stringholder);
// extract the sequence of strings
int strlength = stringholder.value.length;
String [] mystring = stringholder.value;
```

```
for (int I = 0; I ¡ strlength ; i++ )
System.out.println("string" + i + " = " + mystring[i];
```

CORBA provides a mechanism for passing arbitrary data types between distributed components. This mechanism is implemented with the pseudo-object type TypeCode. A pseudo-object type is a type that is used by an ORB implementation when mapping IDL to some programming languages. The type TypeCode is used to describe at runtime the contents of a type any. Java based client applications may insert any user defined data type into the any data type through the use of helper classes which are generated by the IDL compiler and are defined for every user defined type. Applications can interpret at runtime the data type that is included in the type any by invoking the type method on the data type any. The type method returns the typecode of the included type. The following code example illustrates how a sequence of Strings is returned from the PHENIX Naming Service component to a Java application.

```
// IDL file
interface nameserv {
void getNsObjectList( out Any namelist);
};
// Java
// Client.java
org.omg.CORBA.AnyHolder anyholder = new AnyHolder();
Any rtnany = new IE.Iona.OrbixWeb.CORBA.Any(IE.Iona.OrbixWeb_CORBA.IT_ORBIX_OR_KIND);
String[] namelist;
try { nsRef.getNsObjectList( anyholder); } catch (SystemException se) { System.out.println("get
any error" + "Unexpected exception:" + se.toString ()); return; }
// extract the any from the anyHolder
try {
rtnany = anyholder.value;
// test data for a returned sequence.
if ( rtnany.type().kind().value() == TCKind.tk_sequence.value() ) {
namelist = seqStringHelper.extract (rtnany);
// print out the returned list of strings listsize = namelist.length;
for (int index = 0 ; index ¡ listsize ; index++ )
System.out.println("NL["+index+"] = " + namelist[index] ); }
} catch (org.omg.CORBA.TypeCodePackage.BadKind kse )
{ System.out.println(" bad typecode " ); }
} // end sequence
```

The above mechanism has proven to be an easy to use and successful mechanism of passing complex user defined data structures between Java client and C++ server applications. While only simple structures have been shown here, in practice more complicated nested structures containing unbounded sequences of strings, octets and structures have been successfully transmitted.

7 Conclusion

Java has proven to be an effective framework for the development and implementation of control and monitoring applications. It has been successful in allowing application code to be developed and ported seamlessly between different platforms including Solaris, NT and Linux. Security in Java clients remain as much a policy issue and issue in development effort than an intrinsic technical problem. Performance is an area which could be an issue. However, for detector configuration, control and component monitoring, the performance has proven to be adequate.