Efficiency of Distributed Parallel Processing using Java RMI, Sockets, and CORBA

Dr. Roger Eggen
Department of Computer and Information Sciences
University of North Florida
Jacksonville, Florida 32224

Dr. Maurice Eggen
Department of Computer Science
Trinity University 78217

Abstract *Software development is proceeding at a remarkable rate. Many new tools are available to the researcher in parallel and distributed processing. These tools include PVM, MPI, and Java. But, recently, a more general tool, Common Object Request Broker Architecture, (CORBA) has appeared. Since it allows a general view of remote object invocation, and hence another look at distributed parallel programming, CORBA is enjoying considerable attention. In this paper, we consider CORBA and Java, focusing on the overhead and efficiency of Java sockets, Java RMI, and Java's interface to CORBA.*

Keywords sockets, RMI, CORBA, distributed parallel processing

1 Introduction

Computer scientists, along with software engineers, are enjoying a remarkable period of increased processing speeds and declining hardware prices. These new machines present unique challenges to the computing professional to develop software that adequately takes advantage of these computing systems. While techniques and opportunities exist to create sophisticated programs, there is an ever increasing supply of challenging problems requiring ever faster and more capable computers. The software developer can now afford clusters of efficiently networked machines providing new challenges to utilize these parallel distributed systems.

The development of software tools to take advantage of fast new hardware traditionally lags behind hardware production and development. However, new software programming tools have been implemented in an attempt to take advantage of the programming environments. Most notably, the object oriented paradigm through Java is recognized as a major component of web based distributed programming. The Java Development Environment (JDE) includes several features which facilitate the production of stable, robust code for parallel processing. Threads provide an efficient and effective paradigm for utilizing tightly coupled systems and socket communication, remote method invocation (RMI), and the common object request broker architecture (CORBA) provide convenient tools for utilizing distributed systems.

CORBA allows applications implemented in differing languages the ability to communicate while RMI is a Java feature. Sockets provide a lower level of communication where the programmer is responsible for establishing the method of communication. This paper is the result of research studying the efficiency and ease of use of CORBA, RMI, and sockets in a distributed parallel environment.

2 Hardware

The hardware for these experiments consists of a cluster of homogeneous workstations all running RedHat Linux v6.2. The machines are all Intel based PC's consisting of single 500 MHz processor connected by 100 megabit fast ethernet.

3 Software

The software for these experiments was Java (TM) 2 Runtime Environment, Standard Edition (build 1.3.0) and java version 1.2.2 available free from Sun. The CORBA tools are Sun's standard also available from http://java.sun.com.

4 Message Passing Systems

Anytime a distributed cluster of workstations cooperates on the solution of a distributed parallel application, a messaging system is involved. The details of the communication vary between software methods, but fundamentally messaging systems must all operate in a similar manner.

Messaging systems are subject to a common set of requirements in order to provide the capability to perform concurrent execution of processes:

- The code in a process is itself inherently sequential
- A process must be able to send and receive messages
- A send operation should be synchronous
- A receive operation should be asynchronous
- A process should have the ability to perform dynamic task creation and allocation
- Messaging should be able to be created and destroyed dynamically

When a process, say process one, sends a message to a process, process two, whether under explicit control of the programmer not, the following must happen:

- Process one must prepare a send buffer
- Process one must pack the information into the send buffer
- Process one must initiate a send
- Process one sends the buffer
- Process two receives the buffer
- Process two unpacks the information from the received buffer
- Process two performs the appropriate computations on the information
- Process two prepares a send buffer
- Process two packs the new information into the send buffer
- Process two sends the buffer etc.

Whether this procedure is handled explicitly by the programmer, as in sockets or implicitly by the programming environment as in CORBA and RMI, is in some sense irrelevant. The point is that it must be done for messaging to occur.

5 Java Message Passing - Sockets

In Java it is possible to "serialize" a data structure. This means that the data as well as its structure is packaged by the language to be stored on an external device, such as a disk drive, or sent over a network to a remote host. Several common data structures, including arrays and dynamic lists, can be serialized.

To do socket communication in any language the programmer must establish the socket number to communicate through. The server establishes the socket from which it will listen and the client, requesting services from the server then writes to that socket number. Java, through its ObjectInputStream and readObject facilities can send and receive a serialized data structure, essentially packing and unpacking, or "deserialize" the data communicated.

Writing to a socket is much like writing to a file. Normally, when communicating through a socket a programmer must send small units of data, but with serialization Java is able to send complete data structures. To keep the implementations consistent, we did not use this serialization feature, but rather only the features provided by the communication mechanism. Thus, small portions of data were repeatedly written to the socket.

6 Java Message Passing - RMI

Unlike sockets, Java RMI is a higher form of communication where the messaging is handled by the environment rather than explicitly by the programmer. It is the programmer's responsibility to provide the appropriate environment for the remote methods to be invoked. Once the environment has been established, the messaging is handled as a simple method invocation, hence the name.

Java remote method invocation establishes interobject communication. It is fundamental to the Java model. If the particular method happens to be on a remote machine, Java provides the capability to make the remote method invocation appear to the programmer to be the same as if the method is on the local machine. Thus, Java makes remote method invocation transparent to the user.

RMI packages data similar to socket communication described above. Thus, any object, including all the object references, can be sent. This is an extremely powerful feature of RMI, the fact that both the object and the methods for manipulating that object can be sent over a network.

When you invoke a method on a remote machine, the stubs and skeletons are used in the invocation. The stubs and skeletons are local code that serves as a proxy for the remote object. Fortunately, the programmer never has to work with stubs and skeletons directly. The stubs and skeletons are created by running the rmic -- the RMI compiler. After compiling your Java server program and your Java client program, the rmic is executed on the java files that define the interface between the remote methods.

One final piece of the puzzle must be introduced, the RMI registry. Your code must use the registry to look up a reference to the remote object on the remote machine. An analogy may be wishing to send a letter to a friend. Before the letter can be sent, you must look up the address in your address book. RMI must be able to look up the remote object in the registry. The rmiregistry must be running on each of the server machines involved in the application.

7 Object Request Broker - CORBA

CORBA, like Java RMI, was developed so that so that different computer applications could work together over networks. Working with CORBA is similar to working with RMI, Java's own Object Request Broker (ORB). However, CORBA was designed so that any applications adhering to the CORBA standard are able to communicate. CORBA translators are provided for C/C++, Java, Perl, Smalltalk, and even COBOL, to name just a few.

CORBA applications, like Java applications, are composed of objects. Fundamentally, distributed applications require the ability to request the invocation of a remote object, very similar to Java's RMI. CORBA is built around three building blocks:

- The Object Management Group Interface Definition Language, OMG IDL.
- The Object Request Broker (ORB)
- The Internet Inter-ORB Protocol (IIOP)

Each object type must have its interface defined in OMG IDL to indicate how the remote method is going to be invoked and the parameters it requires. The IDL file is at the heart of ability to invoke methods or functions from different languages. This Interface Definition must necessarily be independent of the implementation language of the object. However, each of the popular programming languages have OMG standards by which they will be known. C, C++, Java, etc. all have been defined. The interface is defined very strictly, but the underlying implementation of the object is encapsulated so that it can be implemented in the language chosen by the programmer.

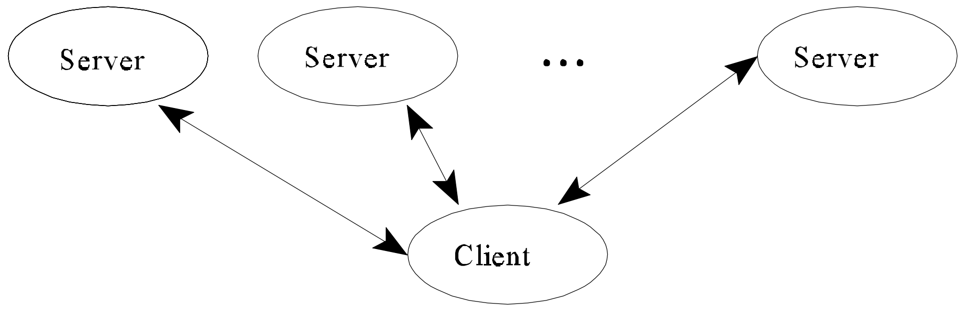
A client requests an object identified by the IDL stub which is the object's proxy in a similar manner as Java's RMI objects are referenced. The ORB handles the request, which is passed along to the server through an IDL skeleton which invokes the object's implementation. In CORBA, every object instance must have its own unique object reference, which clients use to direct their invocations. In this way, different object instances are identified.

We note that in order for the object reference to be passed from client to server and returned, the data is marshalled, in the same way as in the Java RMI model. Thus, the CORBA model can be viewed as a message passing system, as described earlier. The data must be prepared, packed, sent, received by the server, unpacked, manipulated, packed again, and returned to the client.

8 The Results

To evaluate software systems for performing distributed parallel processing, it is important to note that the programs used for the testing should be real applications. For this reason, the authors of this paper have chosen an experiment involving sorting. Since sorting is ubiquitous in many applications, it represents a problem with real world applicability.

Three programs were implemented in a consistent manner utilizing the features provided by each of sockets, RMI, and CORBA. Data was distributed from the client to each of several servers. The servers processed the data and returned the results to the client as shown in Figure 1. In an effort to control the variability of this study the software was designed within the specific features provided by sockets, RMI, and CORBA. The project was implemented in a threaded Java environment, allowing the client to communicate in a timely manner with each of the servers. The research captures the efficiency and ease of programming using sockets, RMI, and CORBA as the underlying communication structure.



Fi
g
ur
e
1

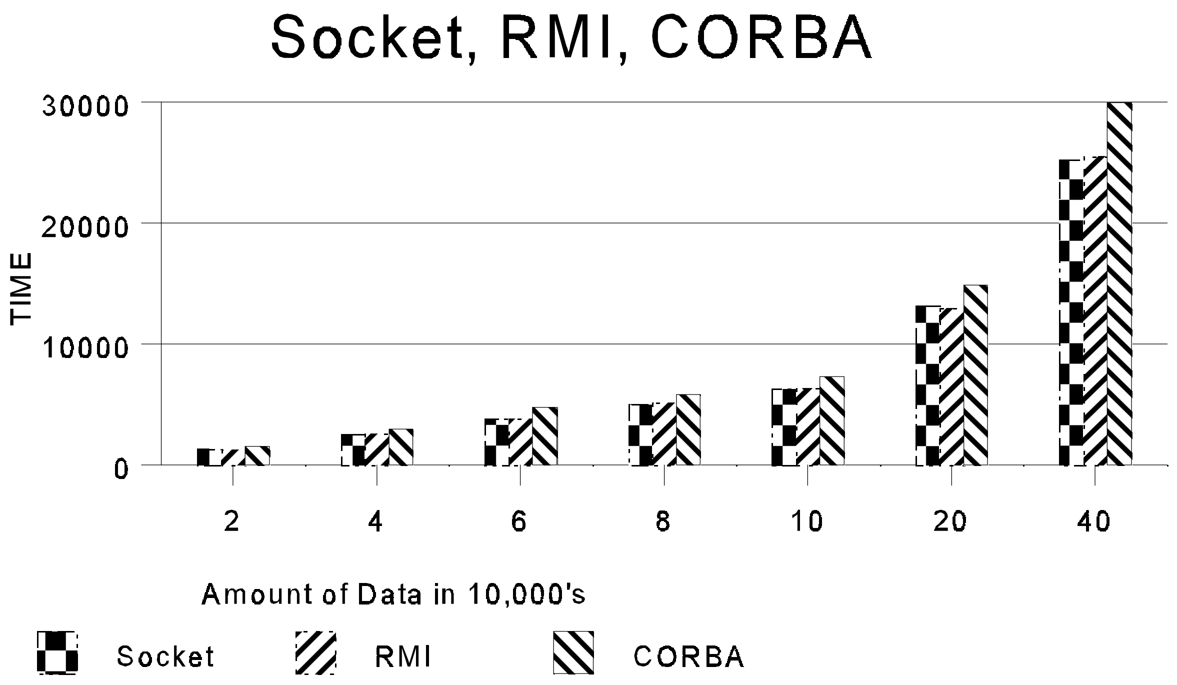


Figure 2

The research evaluates performance of sockets, RMI, and CORBA using 2, 4, and 8 servers over data ranges of 20,000 to 400,000 integers. The servers do all the work while the client distributes and receives data to and from the servers. All tests were executed under exactly the same conditions for each of the communication protocols. The table in Figure 2 summarizes the results.

From Figure 2 and Figure 3 we see the implementations of RMI and sockets provide very nearly the same performance. This is due in part to the way sockets are implemented, if the serialized feature of the Java language had been used, sockets would have been enhanced as evidenced in [4]. CORBA averages 17% longer than RMI or sockets.

COMMUNICATION and COMPUTATION TIME									
no. of	servers	8			4			2	
DATA	socket	RMI	CORBA	socket	RMI	CORBA	socket	RMI	CORBA
20000	1295	1270	1500	2441	1616	1652	4373	2305	2548
40000	2524	2547	2951	4698	3301	3225	8171	4776	5054
60000	3788	3823	4732	7294	4900	4817	12271	7173	7583
80000	4998	5103	5800	9748	6511	6327	16357	9242	10032
100000	6285	6357	7313	12284	8101	6379	23232	11460	12529
200000	13184	12954	14870	24681	15910	16128	46181	23506	24829
400000	25251	25491	29934	49108	31543	32251	95167	46220	49616

Figure 3

9 Summary and Conclusions

It is apparent from examination of the data that CORBA is consistently slower than RMI and sockets.. Java makes programming sockets very easy, but the programmers must concern themselves with establishing a lower level of communication. To effectively use socket communication, the data must be serializable and transferred as a package.

There is slightly more initial work required to set up RMI, but after this is accomplished, using the system is nothing more than a method invocation. For RMI, the rmiregistry must be executing on each of the servers and for CORBA the transient name server need only execute on the network. Both RMI and CORBA require the application to register the remote methods with the name server. There is more to be done to communicate with the ORB in a CORBA application than with RMI. But, CORBA allows the flexibility of applications implemented in different languages to communicate.

For raw speed, one probably would not chose Java in the first place, but for ease of programming, Java is a very good choice. Sockets require more instructions executed each time a send or receive is done. RMI is easy to program and requires a one time communication setup, but is only Java to Java. CORBA is harder to program, but also requires a one time communication setup and provides the opportunity to communicate between applications implemented in different languages. CORBA requires a minor understanding of IDL and experiences a performance hit from this generality.

10 Directions for Future Research

There are several problems to study in this area. It would be interesting to use Java to do the communication between client and server where the server invokes a native method, implemented in C or C++, to do the work. Similarly, the client should be implemented in Java (since Java provides such a nice programming environment) and use CORBA to communicate to a server implemented directly in C or C++. It is expected that these two techniques would yield similar performance characteristics and possibly superior performance to the techniques considered here.

There are other distributed parallel program environments, each with their own advantages and disadvantages, that should be considered. A few of these are LAM (local area machine), PVM (parallel virtual machine), and

MPI (message passing interface). LAM and MPICH are both implementations of MPI, it would be interesting to determine which of these are more efficient.

There are a variety of implementations of the CORBA standard, each requiring a slightly different application programming interface. Some of these are ORBacus, Visibroker, Java2 version 1.3, and OMNIORB. It would be interesting to determine which of these implementations are most efficient and characteristics required for their usage.

11 Bibliography

[1] Holmes, Barry, Programming with Java, Jones and Bartlett, 1998

[2] Wilkinson, Barry and Allen, Michael, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Prentice Hall, 1999

[3] Harold, Elliotte R., Java Network Programming, O'Reilly Publishers, 1997

[4] Eggen, Maurice and Eggen, Roger, "Efficiency of Parallel Java using SMP and Client-Server," Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Volume V, CSREA Press, June, 2000, pp. 2416-2422.

[5] Eggen, Maurice and Eggen, Roger, "Java versus MPI in a Distributed Environment," Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Volume I, CSREA Press, July, 1999, pp 390-395.

[6] Eggen, Roger and Eggen, Maurice, "A Comparison of the Capabilities of PVM, MPI and Java for Distributed Parallel Processing, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Volume I, CSREA Press, July, 1998, pp 237-242.

[7] Hennessy, John and Patterson, David, Computer Organization and Design, Morgan Kaufmann Publishers, 1998.

[8] Lea, Doug, Concurrent Programming in Java, Second Edition, Design Principles and Patterns, Addison Wesley Publishers, 2000.

[9] http://www.sun.com/Java

[10] http://www.omg.org

[11] Begley, Geoff personal contact with Roger Eggen, Summer 2000.

[12] Farley, Jim, Java Distributed Computing, O’Reilly Publishers, 1998

[13] Hortsmann, Cay and Cornell, Gary, Core Java Volume II Advanced Features, Prentice Hall, 2000.