JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 16, 365-379 (2000)

Short Paper

The Design and Implementation of a Multi-Threaded Object Request Broker

WINSTON LO, YUE-SHAN CHANG[+,*]
SHYAN-MING YUAN[+] AND DERON LIANG[++]
Department of Computer and Information Engineering
Tung Hai University
Taichung, Taiwan 407, R.O.C.
[+]*Department of Computer and Information Science*
National Chiao Tung University
Hsinchu, Taiwan 300, R.O.C.
[++]*Institute of Information Science, Academia Sinica*
Taipei, Taiwan 115, RO.C.
[*]*Department of Electronic Engineering*
Ming-Hsin Institute of Technology
Hsinchu, Taiwan 304, R.O.C.

Multi-threaded programming is a well-known technique for improving the performance of applications. In a CORBA environment, clients can invoke shared remote objects. If these objects are single-threaded, the performance of the system in the large distributed applications is affected. This paper presents a detailed description of the design and implementation of a multi-threaded Object Request Broker (ORB) on CORBA. The ORB was implemented on top of Windows NT and the underlying TCP protocol. The system's performance in both one-way and two-way requests is compared with that of a well-known commercial product, the IONA Orbix.

Keywords: object request broker, multi-threaded programming, CORBA, distributed object oriented computing environment, middle-ware

1. INTRODUCTION

With the development of distributed systems and object-oriented technology, object-based distributed processing has become highly valued. Object-oriented design [1, 2] provides high-level abstraction for describing the nature of software components. With its class inherence mechanisms, object-oriented technology also offers greater potential in portability and code reusability. In recent year, some object interconnection standards have been proposed for integrating software objects in a distributed environment; notably, the Object Management Group's (OMG) Object Management Architecture (OMA) [3, 4], Microsoft's COM/DCOM [5], Sun's JAVA Bean [6], etc. OMG's CORBA (Common Object

Received February 10, 1999; revised November 1, 1999; accepted December 22, 1999.
Communicated by Chyi-Nan Chen.

Request Broker Architecture) standard is an open system model, which has already attracted more than one thousand participants in academia and industry. As a result, more and more services [4] and applications have been built based on it.

In the OMA model, an object that provides service to clients over a network is called an object implementation. An ORB (Object Request Broker) serves as a software interconnection bus between clients and object implementations. COSS (Common Object Service Specification) defines several commonly used services in distributed systems, such as transaction service, persistence service, etc. CFA (Common Facility Architecture) specifies a few facilities that are closely connected to the application level and are more like specific application domains, such as common task management tools and facilities for financing and accounting systems.

CORBA is structured so as to allow integration of a wide variety of object systems [7]. Several major components go into constructing CORBA, as shown in Fig. 1. We will now briefly describe the basic concept of CORBA and its components. To invoke an operation on a remote object implementation, a client must first bind to that object. The ORB first checks whether or not the remote object implementation exists. If it does not, an instance of the object implementation is initialized. Binding to a remote object will create a local proxy for the remote object in the client program, and the requests on the local proxy will be delivered to the remote object. The object implementation replies with the results of the invocation to the client via the ORB once the object implementation completes execution of the invocation. The client may send prior requests to the same object implementation without re-establishing the communication channel. The client program can also invoke a remote object implementation via the Dynamic Invocation Interface (DII). That is, the DII allows a client to directly access the underlying request transport mechanisms provided by the ORB core. The DII is useful when an application has no compile-time knowledge of the interface it is accessing. It can query the Interface Repository so as to obtain information about a remote object implementation. Analogously, the Implementation Repository is used by the Object Adapter to find the location of an object server if the target object is not activated.

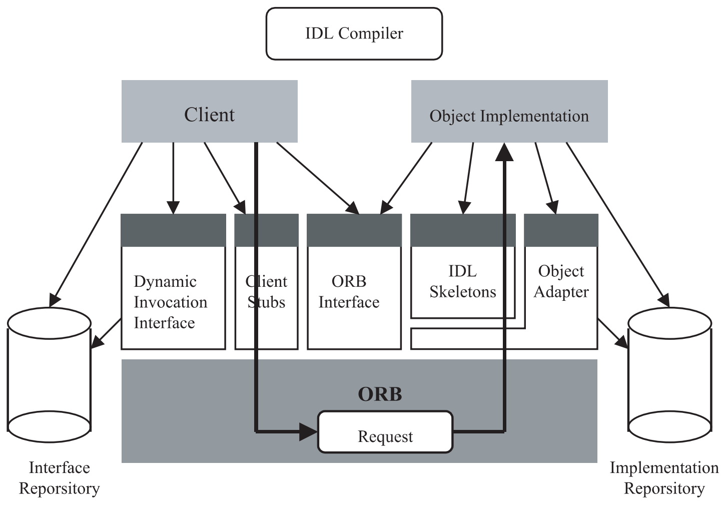

Fig. 1. CORBA architecture.

In addition, multi-threaded programming is a well-known technique for improving the performance of applications. In a CORBA environment, clients can invoke remote objects that are shared. If those objects are single-threaded, this will affect the performance of the system for large distributed applications. Therefore, a multi-threaded ORB is important when developing a large client-server application. The major goal of this study was design and implement a multi-threaded object request broker that is fully compliant with CORBA2. 0. The motivations for this work are as follows: First, it will support related work in our group, such as work on concurrency control service, object transaction service [8], and the heterogeneous full-text search engine agent [9], the aim of which is to develop a multi-threaded version. Second, prior to beginning this work, no commercial multi-threaded version object request brokers were available.

In addition, we have introduced an IDL compiler [10] into the setup to translate IDL definitions into C++ mapping. We have developed the system on top of Windows NT and the underlying TCP transport protocol. In this design, the run-time library plus the server daemon is used to integrate applications and object implementations. The system's user interface was designed to be as similar as possible to Orbix. Finally, the performance of the system has been compared with that of Orbix.

The rest of this paper is organized as follows. Section 2 surveys some well-known products. Section 3 describes the design of the system and of its overall architecture. Section 4 presents the implementation of the system. Section 5 compares the performance results with those of Orbix, and section 6 provides concluding remarks.

2. RELATED WORKS

Here follows a brief description of a few internationally known systems. Most of these systems are commercial products, the detailed architectures of which are usually proprietary to the vendors, and details are not publicly available. Those systems are Electra [11], Orbix [12, 13] and Visibroker [14, 15] etc.

Electra

Electra is not a commercial product but is being used in research projects. It includes the CORBA Static Invocation Interface, ORB Interface, and Basic Object Adapter (BOA), all based on the Dynamic Invocation Interface (DII), which can be seen as the core of the ORB. The core itself is built based on a multicast RPC module supporting asynchronous RPC to both singleton and group destinations. The VOS is a Virtual Operating System layer used by an application to interact with the underlying operating system in a portable and thread-safe way. The VOS mainly provides operations for file and memory management.

Orbix

Orbix is a commercial product from the IONA Technologies Company. It is an ORB that is fully compliant with CORBA. It provides a CORBA environment with the following components: (1) a standard object oriented programming environment, (2) abstract interface definition and its (3) distributed object. The current version of Orbix is 2.0, and it may be ported to more than twenty platforms, such as Unix, Solaris, WindowsNT&95, AIX, OS/2 etc. The Orbix architecture includes a set of libraries that client programs and object servers can link up with and an Orbix Daemon on the server side for accepting requests from client programs. But unlike CDR (Common Data Representation), which is the CORBA 2.0 IIOP (Internet Inter-ORB Protocol) standard, it adopts XDR to encapsulate client request messages.

VisiBroker

VisiBroker is a commercial product from the Visigenic companies. It also is a CORBA-compliant ORB, and it uses IIOP's CDR to encapsulate request messages. With VisiBroker, it is implemented as an Agent-Based structure, and it supports a smart-agent used to manage server objects. The current version of VisiBroker is 3.0 for C++ and Java, and it can be ported to Sun Solaris, HP-UX, IBM AIX, SGI IRIX, Digital UNIX, Windows 95, Windows NT etc.

3. SYSTEM DESIGN

3.1 System Architecture

In CORBA, the ORB is the physical entity that enables a client program to communicate transparently with a server object. It is responsible for delivering requests to the server and the results to the client. The ORB is divided into three layers: a communication layer, a data representation layer and a run-time layer, as shown in Fig. 2. A client program can invoke a remote object either through a client stub or a DII. However, in this system, a client stub also uses a DII to invoke a remote object. An IDL compiler will encapsulate the DII run-time library into a client stub when the client stub and the server skeleton are generated. In this system, the run-time layer is the implementation of the dynamic invocation interface. The data representation layer maps the CORBA Interface Definition Language (IDL) data types into a portable network format. According to the CORBA 2.0 standard, the data representation layer must be in the Common Data Representation (CDR) format. The communication layer is the physical communication channel by means of which a client program and an object server can exchange information. As the system is intended to be executable on the Internet, the TCP/IP protocol is used in this layer.

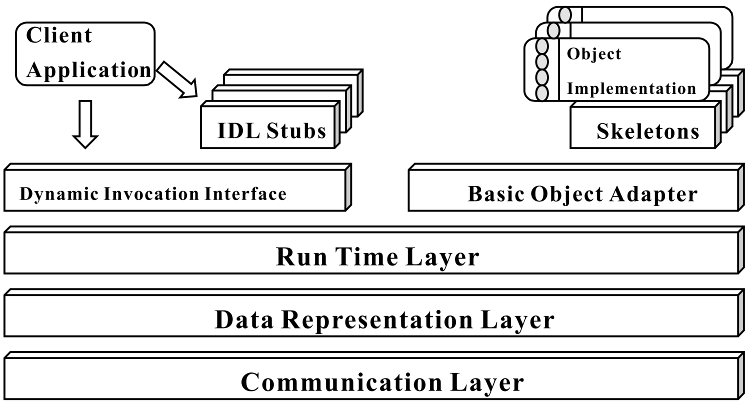

Fig. 2. Our ORB architecture.

The ORB architecture is similar to that of Orbix, which has a set of libraries both on the client side and on the server side. In addition, it has a daemon process on the server side. Application programmers can use the DII to write a client program and link it to the run-time library. On the server side, a daemon that is a process running in the background stands by to receive any requests from the client. The server side architecture is shown in Fig. 3. The daemon process may receive any requests from the network and retrieve some information from request messages. If a requested object is activated, the daemon process can send the related information to the object implementation through the server skeleton. If the requested object is deactivated, the daemon process will query the implementation repository and activate the target object implementation. Then, the daemon process will transfer the request to the target object implementation and obtain a result. Finally, if necessary, it will pass the result back to the client if necessary.

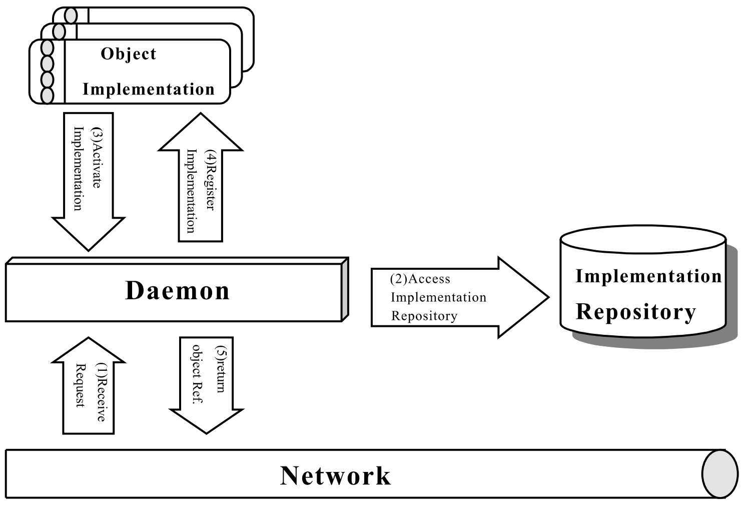
Object
Implementation
(3)Activate Implementation
(4)Register Implementation
Daemon
(2)Access Implementation Repository
Implementation Repository
(1)Receive Request
(5)return object Ref.
Network

Fig. 3. Server side architecture.

3.2 Multiple Threads Support

To handle multiple threads, it is necessary to identify the responsibility in the architecture. The communication layer is the physical communication entity, which only sends and receives messages between the client and the object implementation. The data representation layer maps the IDL data types into the portable network format and vice versa. Therefore, the responsibility for handling multiple threads must lie with the daemon and object implementation. In this design, the BOA is divided into two parts, one being the daemon part and the other the object implementation, so as to handle multiple threads.

The thread invocation scheme in this system, eager invocation, is designed to enhance the system performance. This scheme creates a thread pool to serve incoming requests. If the daemon receives a request, it will allocate a thread to serve the request, and when that is accomplished, it will withdraw the thread.

4. IMPLEMENTATION

This section examines the implementation of the ORB in detail, focusing in particular on multiple thread support.

4.1 IDL Compiler

In this system, an IDL compiler [10] is also implemented. In the IDL to C++ mapping, the translation generates files, including a header file, a client stub and a server skeleton. This is illustrated in Fig. 4. An IDL compiler should generate the following files:

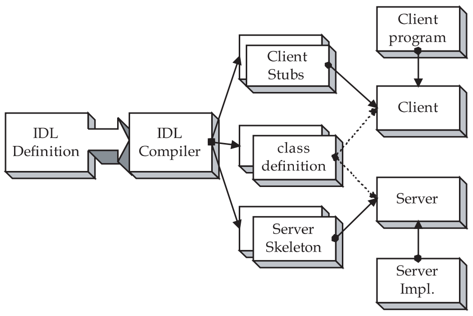

Fig. 4. IDL compiler generated files.

1. Header file: This file includes the class definitions of most data type mappings. IDL data types are purely declarative. Therefore, clients should use the generated data type definitions instead of those defined in the IDL.
2. Client side stub: Each interface defined in the IDL is translated into a corresponding client stub. Subsequent method invocations pass through the stub and are forwarded to the ORB.
3. Server side skeleton: Each interface produces a skeleton for the server. The server skeleton examines an incoming request for this interface and determines which method to use for invocation.
4. Data marshalling and un-marshalling: These parameters require transmission across the network between the client stub and the server skeleton. Therefore, they must be encoded to and decoded from the requests. This is automatically done at the stub and the skeleton.

The IDL compiler is built using general compiler design tools similar to lex and yacc. The lexical analyzer used here is *flex*-the GNU version of lex. It allows accepted tokens to be defined using expressions similar to those of the finite state machine (FSM). The parser generator of choice here is *bison*-the GNU version of yacc. It accepts a BNF grammar specification and produces a LALR(1) parser that accepts language.

The IDL compiler, unlike the general language compiler, does not need to generate binary codes. It only translates IDL declarations into the C++ language. This simplifies the symbol table design for keeping track of the scope names and their corresponding data types. Additional information needed is listed below:

1. Each structured data type needs a flag to distinguish it from a fixed-length type or a variable-length type.
2. Array type definitions need additional size information, which is used in the marshalling routine to determine how many elements it should encode or decode.
3. Constant types need to store values in the symbol table.
4. Enumeration needs information to group members.

The overall structure of the IDL compiler is sketched in Fig. 5. The detailed implementation was given in [10].

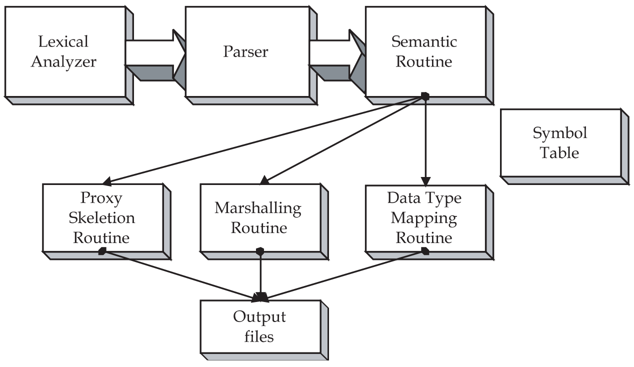

Fig. 5. IDL compiler architecture.

4.2 Server Daemon

On the server side, the ORB is implemented as a daemon and a set of run-time libraries. The responsibility of the daemon is to receive incoming requests from the network. Based on considerations related to multi-threaded design, the daemon process architecture is that shown in Fig. 6. The daemon process receives any request from network and puts it

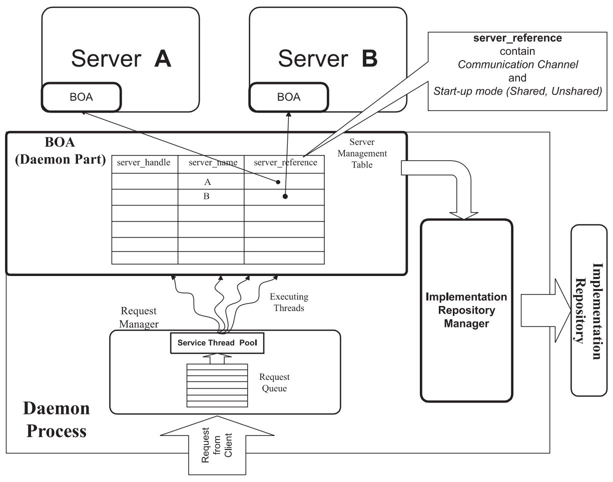

Fig. 6. Daemon process architecture.

into the request queue. To enhance performance, this design uses a thread pool to serve an incoming request. When the daemon is initialized, it creates many service threads in a thread pool. When the daemon receives a request, it allocates a thread to serve the request. The client's requests can be executed concurrently until the number of simultaneous requests exceeds the number of threads in the pool. At this point, additional requests will be queued until a thread becomes available. Once a request is accomplished, the Request Manager withdraws the thread.

The BOA manages the object implementation's life cycle and server activation. In this design, the BOA is divided into two parts: one is the daemon part, and the other is the object implementation part. In the daemon part, the BOA keeps a table that includes the server's handle, name and reference. The server reference contains the communication channel that points to a target server and the start-up mode that indicates whether this server is shared or not while the server name indicates the object implementation.

Another important module in this ORB design is the Implementation Repository Manager. It queries the Implementation Repository if a target object is not activated. The Implementation Repository is a file that contains an object's name, activation mode and the object server location. When a requested object is not activated, the BOA will query the Implementation Repository via the Implementation Repository Manager to get the object server location, and it will then let the object server run.

As shown in Fig. 7, the implementation part of the BOA also contains an Object Description Table (ODT), which contains three fields: an object handle, name and reference. When an object implementation is running, the object manager uses the object reference to point to a target object. The other important module in the object manager is the Server Filter, which is used to filter the target object with or without a multi-threaded version. When the

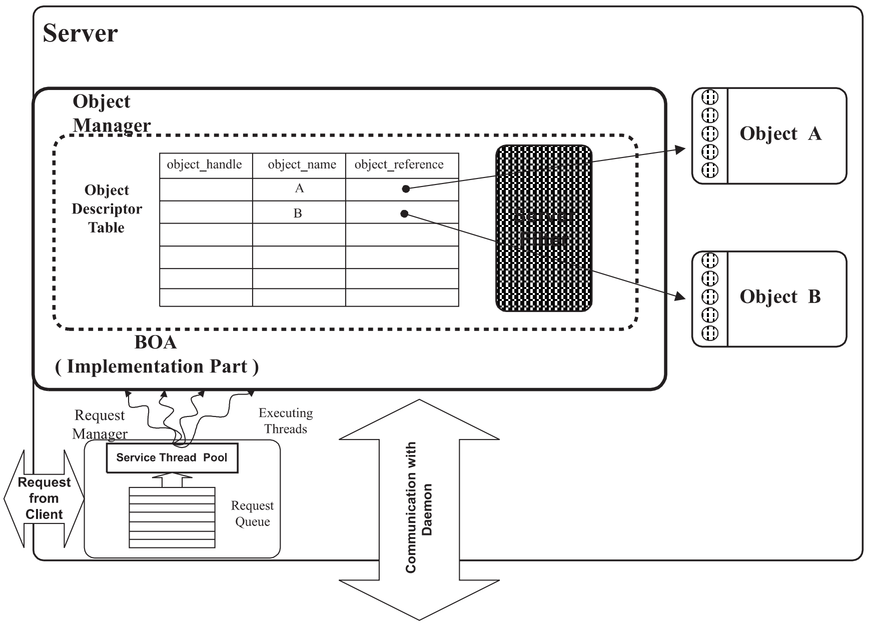

Fig. 7. Object implementation part of BOA.

requested object is not a multi-threaded version, the related information will pass directly to the object dispatcher; otherwise, the BOA will prepare the environment of the multi-threaded version for this request and pass it to the dispatcher. In this design, the object server programmer can write a multi-threaded version object implementation by declaring that the start-up mode is shared and can then put it into the Implementation Repository.

4.3 Dynamic Run-Time Routines.

Run-time routines provide a set of application programming interfaces (APIs) for the client program and the object server. In CORBA 2.0, many pseudo objects must be implemented, such as *Request, Object, NamedValue, NVList, Any, TypeCode, ORB, BOA, ServerRequest* etc. These pseudo objects are necessary for client and object server programming when the dynamic invocation interface is used to invoke an object implementation. Of course, if the client program invokes a remote object via the static interface, the client stub generated by the IDL compiler will cover the scenario of dynamic invocation and use the runtime routines to invoke the remote object. The relationship between run-time routines and applications is shown in Fig. 8. All run-time routines are implemented as the library providing client and server link to it.

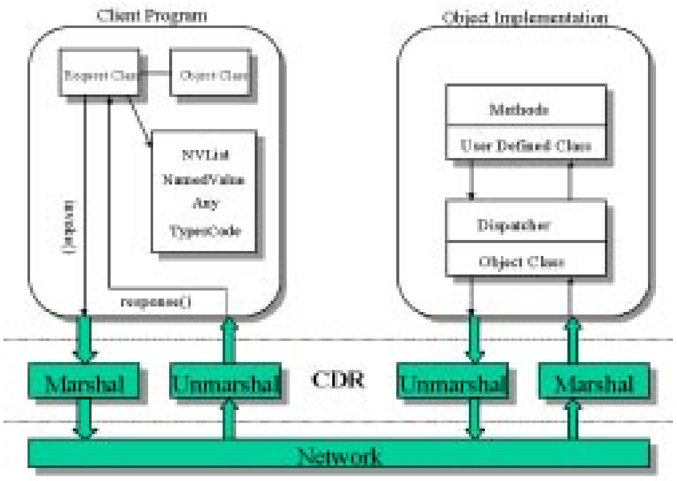

Fig. 8. The relationship between run time routines and applications

When a client invokes a remote object, it must create a *Request* object that inherits from CORBA *Object*. All further operations on the remote object are invoked via this object. The parameters of the invoked operation are placed in the *NVList* object, which is a list of *NamedValues*. Each *NamedValue* is typed as an *Any* that can indicate all types of parameters via the *TypeCode* describing the type of parameter. When the client is ready to invoke an operation on a remote object, it may issue an *invoke()* operation to execute a remote invocation and then execute a *response()* operation to obtain the result from the remote object.

4.4 Data Representation Layer

The data representation layer in the CORBA 2.0 specification is termed the Common Data Representation (CDR). The CDR defines the transfer syntax for transmitting OMG IDL data types across a network. The CDR definition maps the OMG IDL data types from their native host format into a bi-canonical network level representation that supports both the little-endian and big-endian format. The data types provided in the CDR include transferable pseudo objects, such as *TypeCodes*. A CDR encapsulation is a sequence of octets. The CDR is called by run-time routines to encapsulate the information, which includes the invoked object's name, parameters, related data etc., and to create the CDR format.

In this design, the implementation procedure follows CORBA 2.0 and implements two overloading functions, *marshal()* and *unmarshal()*. These functions are used to encapsulate parameters to and draw out parameters from a sequence of octets. This is a non-trivial task in the CDR. To enhance the system performance, C++ in-line functions are employed to implement the *marshal()* and *unmarshal()* function, thus splitting as many as possible large functions into smaller ones duing the implementation. In these two functions, the supported types are char, unsigned *char, short, unsigned short, long, unsigned long, float, double, string, array, NamedValue, NVList, Any, TypeCode* etc. Two more functions, *put()* and *get (),* are implemented for all primitive data types. When a remote object is invoked, the CDR encapsulates all the information from this invocation via the *marshal()* function. The *marshal()* routine then parses all the parameters of the invocation and puts them into the CDR format. Similarly, the server side daemon extracts all the information by executing an *unmarshal ()* operation. This routine parses and obtains data from the CDR format.

4.5 Communication Layer

The responsibility of the communication layer is to move requests and results between clients and servers. In this system, the TCP/IP protocol, which is Win socket in Windows NT, is used to deliver the CDR encapsulation. In this ORB architecture, the communication layer has two parts: the first is the TCP protocol used between the client program and the server daemon, which provides reliable transmission over the network; the second is the Win pipe used between the daemon process and the object implementation [16].

5. PERFORMANCE MEASUREMENT

To evaluate the system performance, Schmidt's method [17] was used to run two tests to compare the performance of our ORB with that of the IONA Orbix Windows Desktop 2.01. One was a one-way paradigm, and the other was a two-way paradigm. These two tests both run under Windows NT on a network connected through 10 Base-T Ethernet.

In evaluating the system performance, the data types used were *char, octet, short, long, double and struct*. These data types were transmitted between the client program and the server object, and the DII transmission performance was measured. The total amount of transferred data was 64Kbytes for each data type, and the data buffers could be adjusted from 128Kbytes to 512Kbytes.

The evaluation results are shown in Figs. 9-12, where the X-axis represents the buffer size of the transmitted data and the Y-axis represents the number of characters transmitted per second. In the one-way paradigm, the two systems had almost the same transmission performance for five primitive data types. Because the buffer size of the socket layer was 8Kbytes and a packet header was added into the transmission buffer when data was transmitted, if the amount of data to be transmitted was 8Kbytes, then the transmission performance was degraded one-way paradigm, as shown in Figs. 9-10. In the case of a two-way paradigm, there was less difference in transmission performance, as shown in Figs. 11-12.

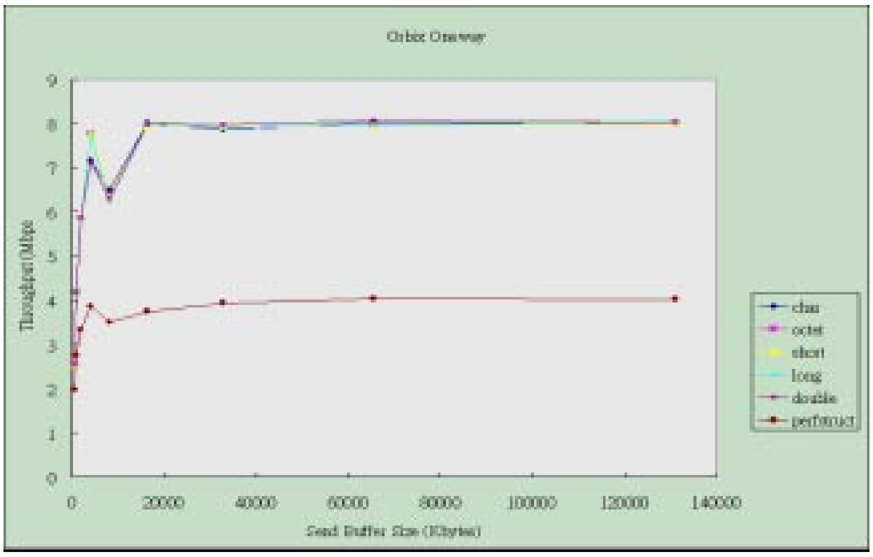

Fig. 9. Orbix one-way transmission result.

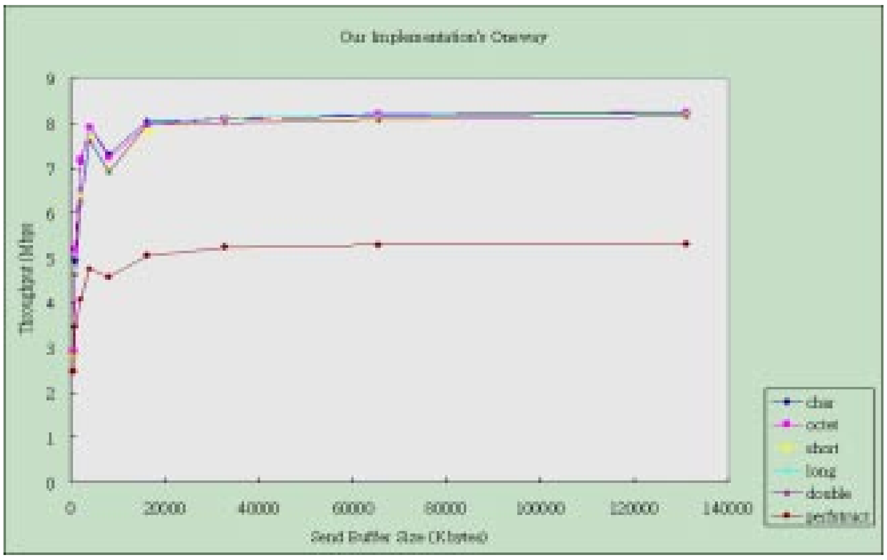

Fig. 10. Our ORB one-way transmission result.

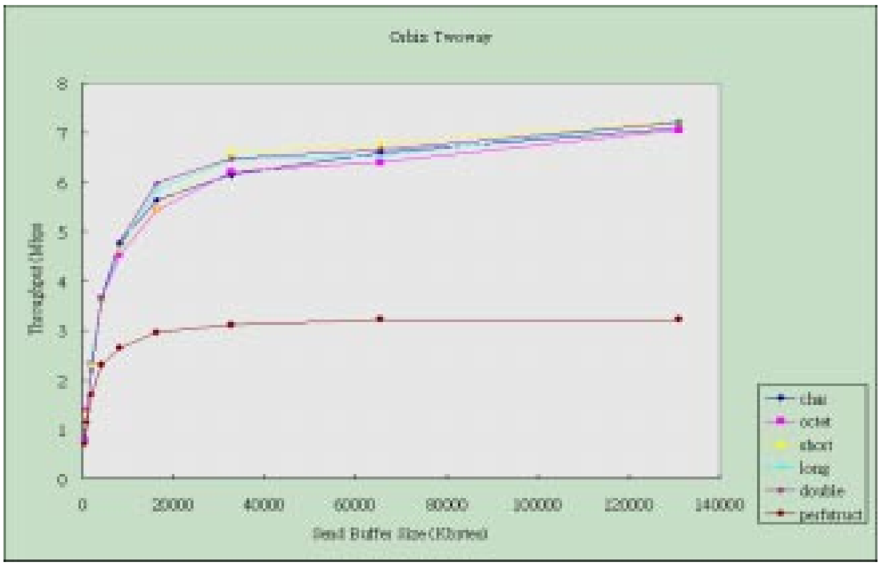

Fig. 11. Orbix two-way transmission result.

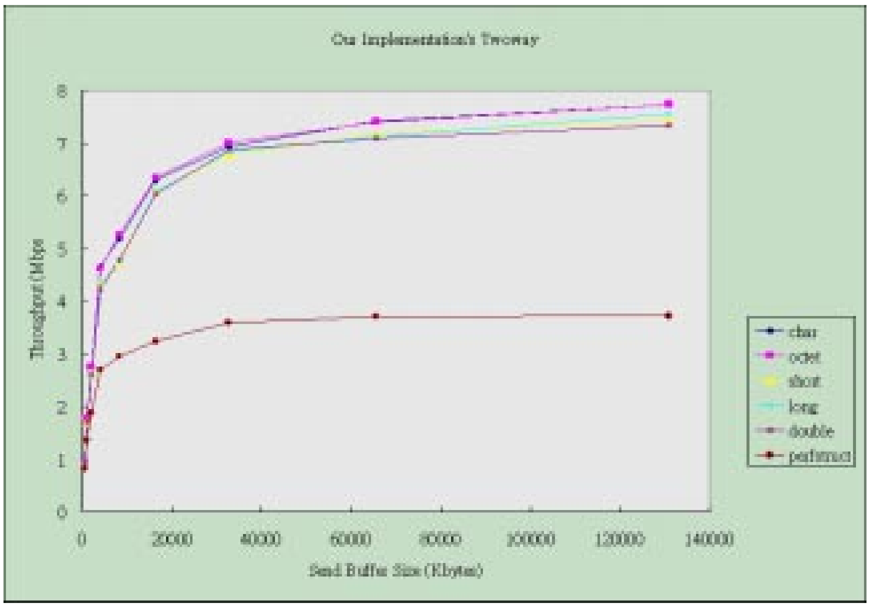

Fig. 12. Our ORB two-way transmission result.

To make comparison with Orbix, DII was used to invoke the server object. The ORB in this design was 8% faster than Orbix for five primitive data types and was 20% faster for the *PerStruct* data type. The reasons for the above results are as follows: First, the eager thread invocation scheme was employed, and a thread pool was created to serve incoming requests. Second, the c++ in-line functions [18] implemented the marshalling and un-marshalling routines to reduce the invocation overhead for small and frequently called methods. Finally, as many as possible large functions, such as the encoder and decoder functions in the CDR, were split into smaller ones [18].

6. CONCLUSIONS

In this study, we employed the CORBA 2.0 to design and implement a multi- threaded object request broker based on Windows NT and the underlying TCP transport protocol. In our design, we divided the ORB into three layers, which included a run-time layer, a data representation layer and a communication layer. The run-time layer is a set of dynamic invocation interface routines which client programs use to invoke a remote object. In the data representation layer, we employ CORBA 2.0, in which the Common Data Representation is used. Lastly, in the communication layer, we use the TCP/IP protocol, which is the standard for the Internet.

In our ORB, we employ a run-time library plus a server daemon to integrate applications and object implementations. The user interface in our system is as similar as possible to that of Orbix because we sought to compare our performance with it. In addition, we implement an IDL compiler in the ORB environment to translate IDL definitions into C++ mapping.

We have compared the performance of our ORB performance with that of the IONA Orbix by employing the Schmit method. The results clearly show that our data marshalling, un-marshalling and optimal memory allocation scheme in the data representation layer are more efficient than those of Orbix. This is because we adopt buffer pool memory management to manage allocated memory.

ACKNOWLEDGEMENTS

The authors thank the referees for their valuable comments. This work was supported by the National Science Council of Taiwan, R.O.C., under Contract No. 86-2213-E-035-005. In addition, we would like to thank Mr. Wang Chii-Jet for his assistance with this project.

REFERENCES

1. G. Booch, *Object-Oriented Design With Applications*, The Benjamin/Cummings Publishing Company, Inc., 1991.
2. G. Booch, *Object-Oriented Analysis and Design With Applications* (2nd Edition), Redwood City, California: Benjamin/ Cummings, 1993.
3. Object Management Group, *CORBA: Common Object Request Broker Architecture and Specification Revision 2.0*, July 1996.
4. Object Management Group, *CORBA Services: Common Object Services Specification*, March 1995.
5. D. Rogerson, *Inside COM – Microsoft's Component Object Model*, Microsoft Press, Redmond, Washington, 1997.
6. Sun Micro, http://www.javasoft.com.
7. A. Ewald and M. Roy, "Choosing between CORBA and DCOM," *Object Magazine*, Vol. 6, No. 8, 1996, pp. 24-30.
8. K. C. Liang, S. M. Yuan, D. Liang, and W. Lo, "Nested transaction and concurrency control services on CORBA," in *Proceedings of Joint International Conference on*

Open Distributed Processing (ICODP) and Distributed Platforms (ICDP), 1997, pp. 27-30.
9. Y. S. Chang, H. C. Hsieh, S. M. Yuan, and W. Lo, "An agent-based search engine based on the internet search service on the CORBA," in *Proceedings of International Symposium on Distributed Objects and Applications (DOA99)*, 1999, pp. 26-33.
10. Y. S. Chang, J. J. Shen, D. Liang, S. M. Yuan, and W. Lo, "An IDL to C++ compiler for the CORBA environment," *The 8th Workshop on Object-Oriented Technology and Applications*, National Central University, Chungli, Taiwan R.O.C., 1997, pp. 91-98.
11. S. Maffeis, "Run-time support for object-oriented distributed programming," Ph.D. dissertration, Dept. of Information Technology, University of Zurich, 1995.
12. IONA Technologies, *Orbix Advanced Programmer's Guide*, 1994.
13. IONA Technologies, *Orbix Programmer's Guide*, 1994.
14. Visigenic Software, http://www.pomoco.com.
15. Visigenic Software, *Programmer's Guide*.
16. A. K. Sinha, *Network Programming in Windows NT*, Addison Wesley, 1996.
17. D. C. Schmidt, "The performance of the CORBA dynamic invocation interface and dynamic skeleton interface over high-speed ATM networks,"*GLOBECOM Conference*, 1996, pp. 50-59.
18. A. Gokhale and D. C. Schmidt, "Principles for optimizing CORBA internet inter-ORB protocol performance," in *Proceedings on Hawaiian International Conference on System Science*, 1998, pp. 376-385.

Win-Tsung Lo (羅文聰) received the B.S. and M.S. degrees in applied mathematics from National Tsing Hua University, Taiwan, Republic of China, and M.S. and Ph.D. degree in computer science from the University of Maryland. He is now an associate professor of computer science and the director of Computer Center at Tung Hai University, Taiwan, Republic of China. His research interests include architecture of distributed systems, data exchange in heterogeneous environments, and multicast routing in computer networks.

Yue-Shan Chang (張玉山) was born on August 4, 1965 in Tainan, Taiwan, Republic of China. He received his B.S. degree in Electronic Technology from National Taiwan Institute of Technology in 1990 and his M.S. degree in Electrical Engineering from the National Cheng Kung University in 1992. Currently, he is a graduate student of PhD. degree in Computer and Information Science at National Chiao Tung University. His research interests are in distributed systems, object oriented programming, fault tolerant, computer network.

Shyan-Ming Yuan (袁賢銘) was born on July 11, 1959 in Mauli, Taiwan, Republic of China. He received the B.S.E.E. degree from National Taiwan University in 1981, the M.S. degree in computer science from University of Maryland Baltimore County in 1985, and the Ph.D. degree in computer science from University of Maryland College Park in 1989. Dr. Yuan joined the Electronics Research and Service Organization, Industrial Technology Research Institute as a Research Member in Oct. 1989. Since September 1990, he had been an associate professor at the Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan. He became a professor in June, 1995. His current research interests include distributed objects, Internet technologies, and software system integration. Dr. Yuan is a member of ACM and IEEE.

Deron Liang (梁德容) received a B.S. degree in electrical engineering from National Taiwan University in 1983, and an M.S. and a Ph.D. in computer science from the University of Maryland at College Park in 1991 and 1992 respectively. He is an associate research fellow with the Institute of Information Science, Academia Sinica, Taipei, Taiwan, Republic of China, where he was an assistant research fellow from 1993 to 1999. Since 1998, he has been jointly appointed with Department of Computer Science, National Taiwan Ocean University as an associate professor.

Dr. Liang's current research interests are in the areas of distributed fault-tolerant software systems, object-oriented technology, and system reliability analysis. Dr. Liang is a member of ACM and IEEE.