

PERFORMANCE EVALUATION OF MIDDLEWARE BRIDGING TECHNOLOGIES

Rod Fatoohi, Vandana Gunwani, Qi Wang & Charlton Zheng

Computer Engineering, San Jose State U., San Jose, CA, Email: rfatoohi@email.sjsu.edu

ABSTRACT

This paper provides a state-of-the-art study of bridging between different middleware technologies. Two DCOM-CORBA bridges, IONA OrbixCOMet and Visual Edge ObjectBridge, as well as a DCE-CORBA bridge by Inprise are tested and evaluated. Several configurations, depending on the number of machines and location of the bridge, are employed and two languages (C++ and Java) are used. The results show that the three bridges perform reasonably well for different configurations and language mappings.

I. INTRODUCTION

A distributed computing system consists of a number of autonomous machines connected by some underlying communication mechanism. The machines in a distributed system may have different hardware architectures and operating systems. Middleware technologies exist to mask the underlying differences that may exist in a distributed system. In today's market, there are several middleware technologies, including: the Distributed Computing Environment (DCE) by The Open Group, the Common Object Request Broker Architecture (CORBA) by the Object Management Group (OMG), the Distributed Component Object Model (DCOM) by Microsoft, and Java by Sun. However, the underlying middleware components are incompatible; and therefore, a higher-level heterogeneity problem has been created.

Recently, software *bridges* have emerged that enable interoperability between different middleware environments. Basically, these bridges are processes that enable communication between clients in one middleware domain and servers in another domain. However, there is no comprehensive study of the functionality and performance of different bridging products. In this study, we focus on three middleware technologies: DCE, CORBA, and DCOM, which are briefly described below; for more details, see also Orfali et al. [8].

DCE [10] is a software infrastructure for developing distributed systems, produced by the Open Group. DCE consists of a set of application programming interfaces and a set of run-time services. It is based on the RPC paradigm. In addition to its RPC, it provides a set of basic services that includes a naming service, a security and authentication service, and a time synchronization service. DCE uses an Interface Definition Language (IDL) to specify procedure interfaces. DCE IDL is similar to other

interface languages, including CORBA IDL, but it is unique (there is no standard for IDLs). Currently, only the C language is directly supported by DCE IDL.

Object-Oriented DCE (OODCE) [3] is an object-oriented wrapper of DCE developed by HP to support an object-oriented development paradigm. In the OODCE development model, C++ is used as the primary language. OODCE uses the DCE IDL to define the server interface. It also uses an enhanced version of the DCE compiler, called *idl++*, to process the interface specification.

CORBA [8] is a standard for transparent communication between application objects. The CORBA specification is being developed by OMG. CORBA Object Request Broker (ORB) enables clients and objects to communicate in a heterogeneous distributed environment. Using an ORB, a client can invoke a method on a server object transparently with respect to its location (local or remote) and its language implementation. The CORBA specification describes the interfaces and services that ORBs must have. The specification includes an IDL for describing interfaces. IDL mappings to several languages (such as C, C++, Java, Smalltalk, Ada, and COBOL) have been specified and provided by many vendors.

Distributed COM [1] is an extension of the Component Object Model (COM), developed by Microsoft Corp. as an object-based programming model for developing and deploying software components. COM is the basis of many Microsoft object-based technologies such as ActiveX and OLE. COM has been part of MS Windows for some time. It separates object interfaces from their implementations, similarly to CORBA. COM defines a binary call standard for interfaces. It also defines a language for specifying interfaces called Microsoft IDL (MIDL). An MIDL compiler generates client proxies and server stubs in C, C++, or Java from an interface definition. The DCOM extension enables COM processes to run on different machines. It includes communication with remote objects, location transparency, and an interface to distributed security services.

This paper presents the results of experiments to evaluate the performance of several middleware bridging technologies. It starts with an overview of bridging technologies, followed by a brief description of two test problems. Next, the results of testing two DCOM-CORBA bridges and a DCE-CORBA bridge are presented and analyzed. Finally, we offer some concluding remarks.

II. BRIDGING TECHNOLOGY

In order to transparently couple components from different middleware technologies, some form of bridging software is needed. A bridge is therefore a process that allows a client in one middleware domain to make requests to, and receive replies from, a server in another middleware domain. The level of communication is determined by what kind of bridge is in use. A bridge possesses several properties, including unidirectional or bi-directional, static or dynamic, customized or commercial and other middleware-specific properties.

A. Unidirectional vs. Bi-directional

Unidirectional means that data can go in one direction only, either from the client to the server or from the server to the client. In order for data to go freely in either direction a bi-directional bridge is required. OMG coined the word inter-working to describe this behavior. This two-way interoperability allows function to pass object and to return object. Combining two unidirectional bridges can not accomplish the same deed since there is only one function call involved.

B. Static vs. Dynamic

A static bridge requires static proxy and stub implementation to perform marshalling for each interface. If there is any change to the interface, the proxy and stub must also reflect this change, thus the interface has to be recompiled to generate new proxy and stub codes to encode, decode and invoke the new interface. On the other hand, a dynamic bridge frees the user from this recompilation task since it has a generic proxy that is capable of marshalling all data types based on some form of a run-time type library. This type of information look-up can be costly in performance. Generally, dynamic bridges are not as efficient as static bridges, but with an optimization like caching, dynamic bridges can catch up to static bridges in performance. Despite this potential performance factor, the generic mapping of the dynamic bridge does offer a smaller memory footprint than the interface-specific mapping of the static bridge.

Yet a third kind of bridges is on-demand bridges. Here a bridge factory automatically generates the source code for a static bridge, usually based on the interface definitions of the client and server, which are to be bridged. Based on these definitions, bridging code must translate values between the two middleware domains, the mapping rules for the respective types are coded into the bridge factory.

C. Customized vs. Commercial

Customized bridge is any bridge that is not offered off the shelf. Normally, customized bridges are not recommended unless performance is critical or third party support is to be avoided. Otherwise, commercial bridge is a better choice

for lower cost in design, implementation, and maintenance. Nevertheless, if one must write his or her own bridge, the easiest approach is to create a Java-based bridge that takes advantage of the built-in support for COM in Microsoft's Java Virtual Machine, for example.

D. ORB-specific vs. ORB-neutral

This property is specific to CORBA but it is applicable to other middleware bridges. An ORB-specific bridge works only with the ORB products of the vendor who supplies the bridge. On the other hand, an ORB-neutral bridge can work with different ORBs.

III. TEST PROBLEMS

Two simple applications were developed to analyze the performance of different bridges. The first application, *count*, was designed to do a "ping" operation from the client to the server. In this application, which is similar to the *Count* program in [8], the client invokes the increment method one thousand times and measures the average time it takes. The second application, *BankATM*, was designed to simulate a "bank" application. The performance of *count* was analyzed both with and without the bridges for comparison.

IV. DCOM/CORBA BRIDGES

A. Overview

Objects using CORBA do not inter-work with objects that are based on the DCOM architecture, and vice versa. To meet this challenge, the OMG has created the DCOM/CORBA Inter-working specification [2, 6, 9]. This specification has two parts: Part A and Part B. Part A defines the bi-directional mapping between non-distributed COM and CORBA. Part B extends the adopted Part A specification to allow inter-working of DCOM and CORBA. As of CORBA 2.3, the inter-working specification has been incorporate into CORBA Architecture and Specification. The inter-working specification (Part A and Part B) specifies six configurations of the bridge, depending on the location of the bridge: on the client machine, on the server machine, or on a third machine [6].

The location of the bridge determines the communication protocol between the client and the server. For example, when the DCOM client talks to the remote CORBA server and the bridge is on the client side, IIOP (Internet Inter-ORB Protocol) is used for communication. On the other hand, when the CORBA client talks to the remote DCOM server and the bridge is on the client side, Object RPC is used for communication. However, in practice, other factors determine the location of the bridge [9]. Housing the bridge with the client means that the actual bridging computation takes place on the client machine. It also

implies that the inter-working software has to be installed on each client machine. Installing the bridge at the server side, on the other hand, means that the bridging computation moves to the server machine, where CPU cycles are in high demand. A third option is installed the bridge on a third machine. The disadvantage here is an additional process needs to be configured and managed.

B. Implementation and Results

Two DCOM-CORBA bridges were considered in this study: ObjectBridge by Visual Edge [11] and OrbixCOMet by IONA Technologies [6]. Both bridges are bi-directional and dynamic. ObjectBridge is also ORB-neutral while OrbixCOMet works with IONA Orbix3 only. Both of them are supposed to be compliant with the Inter-working Specification; however, we found that Objectbridge might lack support for two of the specification configurations.

The *count* program was used to measure the performance of the two bridges. Each bridge was tested for all six configurations mentioned in the inter-working specification. The performance was measured with and without the bridges for comparison. Also, the impact of the language binding was considered. The *count* program was implemented with both C++ and Java for ObjectBridge. In addition, the client was also implemented with Java Applet to measure performance for Web-based applications. In all cases, the same language was used for both the client and the server.

The DCOM-CORBA experiments were conducted using Windows NT 4.0 with Service Pack 5 running on Pentium II machines with 233 MHz and 128 Mbytes of memory. Several software tools were needed for this study, including: ObjectBrident Enterprise Client version 1.1, VisiBroker for C++ version 3.3, VisiBroker for Java version 3.4, Orbix3, OrbixCOMet, OrbixWeb version 3.1, MS Visual C++ version 6.0, MS Visual J++ version 6.0, SDK for Java version 4.0, DCOM Configuration, OLE-COM Object View, and SUN Java JDK 1.1.8 and JDK 1.2.2.

Tables 1 through 4 show the results for different configurations and language mappings. All data were collected when the network (Ethernet) is observed to be idle, usually at night. During performing the experiments, it was observed that network traffic could greatly affect the results. For all these tables, the following notation is used: C: Client, S: Server, B: Bridge and X, Y and Z are three machines.

C. Observations

The results for OrbixCOMet using Java were not available due to many obstacles in integrating Microsoft SDK for Java 4.0 with OrbixCOMet. Unlike ObjectBridge, OrbixCOMet does not work well with Microsoft SDK for

Java. OrbixCOMet does not generate all the information that Microsoft SDK needs, and vice versa. There are many missing pieces that make this task hard to accomplish.

	CORBA S	DCOM S
C O R B A C	C & S on X: 2.470	C, B & S on X: 250.80
	C on X, S on Y: 2.664	C & B on X, S on Y: 250.87
		C on X, S & B on Y: 250.87
		C on X, B on Y, S on Z: 251.73
D C O M C	C, B & S on X: 3.021	C & S on X: 0.23
	C & B on X, S on Y: 3.164	
	C on X, B & S on Y: 3.899	C on X, S on Y: 1.262
	C on X, B on Y, S on Z: 4.183	

Table 1. Results (in ms) of OrbixCOMet C++ Server and Client (C: Client, S: Server, B: Bridge; X, Y, Z: three machines).

The DCOM-CORBA results for C++ using both bridges (Tables 1 and 2) show that ObjectBridge is significantly faster than OrbixCOMet (by a factor of over three for most cases). In one of the configurations (CORBA client with DCOM server), the performance of OrbixCOMet is quite poor due to its current implementation. However, this configuration is not commonly used.

	CORBA S	DCOM S
C O R B A C	C & S on X: 0.397	C, B & S on X: 1.525
	C on X, S on Y: 0.991	C & B on X, S on Y: 1.604
		C on X, S & B on Y: 1.702
		C on X, B on Y, S on Z: 1.834
D C O M C	C, B & S on X: 0.841	C & S on X: 0.23
	C & B on X, S on Y: 1.141	
	C on X, B & S on Y: 1.082	C on X, S on Y: 1.262
	C on X, B on Y, S on Z: 1.191	

Table 2. Results (in ms) of ObjectBridge C++ Server and Client

The single machine implementation shows that both bridges add an overhead of about 0.5 ms each (3.021 ms with the bridge compared to 2.664 ms without the bridge for OrbixCOMet, 0.841 ms with the bridge compared to 0.397 ms without the bridge for ObjectBridge). This is in comparison with CORBA to CORBA communication. In comparison with COM to COM communication, the bridge overhead is even higher.

	CORBA S	DCOM S
C O R B A	C & S on X: 1.432	C, B & S on X: 2.113
	C on X, S on Y: 1.602	C & B on X, S on Y: 1.772
		C on X, S & B on Y: 1.842
C	C on X, B on Y, S on Z: 1.983	
D C O M	C, B & S on X: 0.982	C & S on X: 0.091
	C & B on X, S on Y: 1.332	
	C on X, B & S on Y: 1.352	C on X, S on Y: 1.922
	C on X, B on Y, S on Z: 1.402	

Table 3. Results (in ms) of ObjectBridge Java Server and Client

	CORBA S	DCOM S
C O R B A	C & S on X: 1.573	C, B & S on X: 2.383
	C on X, S on Y: 22.342	C & B on X, S on Y: 2.073
		C on X, S & B on Y: 22.451
C	C on X, B on Y, S on Z: 22.885	
D C O M	C, B & S on X: 1.031	C & S on X: 0.401
	C & B on X, S on Y: 1.372	
	C on X, B & S on Y: 1.392	C on X, S on Y: 1.973
	C on X, B on Y, S on Z: 1.472	

Table 4. Results (in ms) of ObjectBridge Java Server and Applet Client

The results also show that Java performance is lower than C++ performance using ObjectBridge (as expected!). However, for COM to COM communication (with no bridging) on a single machine, Java is faster than C++. The Java COM server is an in-process server, housed in a DLL file while the C++ server is a local server (out-of-process) housed in an EXE file. But when an in-process server runs between machines, it has to run on top of a surrogate process, behaving more like an out-of-process server. Therefore, when the COM client and COM server are running on different machines, C++ outperforms Java.

The timing results for a DCOM client interacting with a CORBA server for multiple machine configurations are about the same regardless to the location of the bridge. These results are about up to 40% higher than the single machine configuration due to network overhead. Within the three different locations of the bridge (on the client machine, on the server machine, and on a third machine), the differences are insignificant.

Another observation is that the VisiBroker Java applet is very slow when the client and server are not running on the same machine. VisiBroker requires Gatekeeper to be activated before an applet can run. Gatekeeper performs security check when the server and client are not running on the same machine.

V. DCE/CORBA BRIDGE

A. Overview

The DCE-CORBA Bridge is a commercial product developed by Inprise Corp. [4]. It can be used to enable CORBA clients to use IIOP to access data and transactions from DCE applications. The bridge acts as a standard CORBA server when accessed by CORBA applications. Bridge objects are CORBA objects that represent DCE servers. These objects are the means by which CORBA clients invoke operations on DCE servers. The bridge acts as a standard DCE client when talking to any DCE server. Thus, bridge objects act simultaneously as CORBA servers and DCE clients. In its role as a CORBA server, the bridge encapsulates DCE systems in an object wrapper that enables the object-oriented CORBA clients to access DCE data and transactions using object semantics.

The Inprise DCE-CORBA bridge is a unidirectional, on-demand bridge. Only CORBA clients can access the DCE servers and not vice versa. The bridge is also an ORB-specific: it supports the VisiBroker ORB only. The bridge automatically generates a CORBA IDL file that maps to the DCE IDL. The generated CORBA IDL contains a CORBA object, called the bridge object, mapped to each DCE interface. The CORBA IDL can then be compiled with a CORBA IDL compiler to generate the client-side

stubs and server-side skeletons. A CORBA client can then invoke methods on the bridge object.

The DCE-CORBA bridge can be used with standalone CORBA applications as clients or even with Web applications. In the case of Web applications, the CORBA client will be a Web browser with a Java applet embedded in the HTML page. The VisiBroker Gatekeeper is used to tunnel the HTTP requests [4].

B. Implementation and Results

The *count* program was run for different configurations with and without the bridge for comparison. When running without the bridge, the program uses the DCE RPC mechanism in one case (DCE client, DCE server) and the CORBA ORB (CORBA client, CORBA server) in another. When the bridge is used, the client is a CORBA client and the server is a DCE server. The CORBA client accesses the bridge using the CORBA ORB and then the bridge invokes the DCE server using the RPC mechanism.

	DCE S	CORBA S
DCE C	C & S on X: 0.698	N/A
	C on X, S on Y: 1.089	N/A
C++ CORBA C	C, B & S on X: 7.832	C & S (C++) on X: 0.583
	C & B on X, S on Y: 7.996	
	C on X, B & S on Y: 7.856	C on X, S (C++) on Y: 0.842
	C on X, B on Y, S on Z: 7.475	
Java CORBA C	C, B & S on X: 8.334	C & S (Java) on X: 1.472
	C & B on X, S on Y: 8.271	
	C on X, B & S on Y: 7.569	C on X, S (Java) on Y: 1.401
	C on X, B on Y, S on Z: 7.768	
Java applet CORBA CI	C, B & S on X: 11.326	C & S (Java) on X: 4.373
	C & B on X, S on Y: 10.909	
	C on X, B & S on Y: 28.161	C on X, S (Java) on Y: 25.807
	C on X, B on Y, S on Z: 29.661	

Table 5. Results (in ms) of DCE-CORBA Bridge

The DCE/CORBA experiments were conducted using Solaris 2.5.1 running on UltraSPARC Iii machines with 333 MHz and 128 Mbytes of memory. Several software tools were needed for this study, including: Transarc DCE

1.1 for Solaris, VisiBroker for Java version 3.3, VisiBroker for C++ version 3.3, VisiBroker Gatekeeper, JDK 1.1.8 and GNU C and C++ compilers.

Tables 5 and 6 show the results for DCE-CORBA bridging for both DCE and OODCE, respectively, using different configurations. The DCE server was written in C while the OODCE server was written in C++. When a CORBA client talks a CORBA server, both sides use the same language. All data were collected while the network (Ethernet) was idle.

	OODCE S	CORBA S
OODCE C	C & S on X: 0.797	N/A
	C on X, S on Y: 1.127	N/A
C++ CORBA C	C, B & S on X: 7.461	C & S (C++) on X: 0.583
	C & B on X, S on Y: 8.041	
	C on X, B & S on Y: 7.510	C on X, S (C++) on Y: 0.842
	C on X, B on Y, S on Z: 7.805	
Java CORBA C	C, B & S on X: 7.847	C & S (Java) on X: 1.472
	C & B on X, S on Y: 7.973	
	C on X, B & S on Y: 7.626	C on X, S (Java) on Y: 1.401
	C on X, B on Y, S on Z: 8.083	

Table 6. Results of OODCE-CORBA Bridge

C. Observations

The DCE timing results (Table 5) show that when the C++ CORBA client, bridge and DCE server are run on the same machine, the average time for the client to ping the server is 7.832 ms. This contrasts with an average time of 0.583 ms when a C++ CORBA client pings a C++ CORBA server. When a DCE client pings a DCE server, the average time was observed to be only 0.698 ms. Therefore, the extra overhead of around 7 ms incurred when connecting a CORBA client to the DCE server is attributable to the delay through the bridge.

When the client pings the server running on another machine but without the bridge, the average time increases due to network overhead. In the case of a C++ CORBA client pinging a C++ CORBA server, the average time rises to 0.842 ms from 0.583 ms in the single machine case. In the case of a DCE client pinging a remote DCE server, the average time is 1.089 ms as compared to 0.698 ms in the single machine case. These results show that the

CORBA communication (IIOP) is slightly faster than the DCE communication (RPC).

When comparing the C++ CORBA client to the Java CORBA client, we found that the average time to ping the DCE server rises from 7.832 ms to 8.334 ms. This indicates that the Java implementation is somewhat slower than the C++ implementation. This is also indicated by the fact that the average ping time for a Java CORBA client to Java CORBA server is 1.472 ms as opposed to 0.583 ms for the equivalent C++ case. When a Java applet is used as the CORBA client, the performance is significantly slower. The average ping time for a Java applet client to a DCE server is 11.326 ms and is 4.373 ms for a Java applet client to a Java CORBA server. The extra delay for the applet client can be attributed to the VisiBroker Gatekeeper, which is used to tunnel the HTTP request to the server.

The average times for a C++ CORBA client to ping a DCE server through the bridge and across the network range between 7.475 ms and 7.996 ms for the multiple machine configurations. Interestingly, these times are not much different from the value of 7.832 ms observed for the single machine case. This can be due to two reasons: 1) The time to transmit over the network is a small fraction of the overall time to connect the client and server through the bridge and 2) With the client running on one machine and server on another, the CPU workload on both machines is lower compared to the single machine case and this may offset some of the overhead incurred by using the network.

Similarly, for the Java CORBA client to ping the DCE server, the average times range between 7.569 ms and 8.271 ms for the multiple machine configurations compared to 8.334 ms for the single machine configuration. Java applications are known to be CPU intensive and hence, the second reason given above may be even more applicable in this situation. The average ping time for the Java applet and DCE server case increases from about 11 ms for the single machine case to over 28 ms for two other configurations. The extra overhead is possibly due again to the VisiBroker Gatekeeper.

The OODCE timing results (Table 6) show that the performance of the bridge with DCE or OODCE on the server side is about the same (within about 5% difference). This shows that OODCE does not add a major overhead to DCE.

VI. CONCLUDING REMARKS

In summary, the three bridges (Visual Edge ObjectBridge, IONA OrbixCOMet, and Inprise DCE-CORBA bridge) performed well using the two simple applications: *count* and *BankATM*. The *BankATM* application was used mainly to verify the proper operations of the bridges. Based on the

results of the *count* program, ObjectBridge has a performance advantage over OrbixCOMet. However, due to the simplicity of the program, a more complex application should be employed to further verify this conclusion. Despite the performance issue, OrbixCOMet offers more bridging configuration and better documentation support.

The overhead incurred in using the Inprise DCE-CORBA bridge is approximately a few milliseconds for various client/server configurations. This is an acceptable overhead when leveraging legacy applications implemented using DCE or OODCE. An existing DCE framework can be utilized without any modification to the server code or the deployed configuration. However, unlike the two DCOM-CORBA bridges, the DCE-CORBA bridge is a unidirectional and on-demand bridge.

The three bridges performed reasonably well for the most common configurations (DCOM client with CORBA server and CORBA client with DCE server). The location of each bridge (close to the client, close to the server, and on a third machine) seems to have a little impact on the performance. In general, C++ implementations perform better than Java implementations and significantly better than Java applet implementations.

VII. ACKNOWLEDGEMENT

We would like to thank Inprise, Visual Edge and IONA technologies for providing the three bridges for testing and evaluation.

VIII. REFERENCES

1. Eddon, G., and Eddon, H., 1998. *Inside Distributed COM*, Microsoft Press.
2. Geraphy, R., Joyce, S., Moriarty, T., and Noone, G., 1999. *COM-CORBA Interoperability*, Prentice Hall.
3. HP, 1994. *HP Object Oriented DCE C++ Class Library*.
4. Inprise, 1998. *Inprise DCE-CORBA Bridge User's Guide*.
5. IONA, 1998. *OrbixCOMet Desktop Programmer's Guide and References*.
6. OMG, 1997. *COM/CORBA Interworking Part A & B*.
7. Orfali, R., and Harkey, 1998. *Client/Server Programming with Java and CORBA*, 2nd ed., Wiley.
8. Orfali, R., Harkey, D., and Edwards, J., 1999. *The Essential Client/Server Survival Guide*, 3rd ed., Wiley.
9. Rosen, M., and Curtis, D., 1998. *Integrating CORBA and COM Applications*, Wiley.
10. Rosenberry, W., Kenney, D., and Fisher, G., 1992. *Understanding DCE*, O'Reilly & Associates, Inc.
11. Visual Edge, 1998. *ObjectBridge: The Premier OLE/COM-CORBA Interoperability Solution*.