Improving the Performance of Distributed CORBA Applications

Shivakant Mishra
Department of Computer Science
University of Colorado, Campus Box 0430
Boulder, CO 80309-0430, USA.
mishras@cs.colorado.edu

Nija Shi
Department of Computer Science
University of Wyoming, P.O. Box 3682
Laramie, WY 82071-3682, USA.
nijashi@yahoo.com

Abstract

This paper proposes a new technique called CORBA-as-needed to improve the performance of distributed CORBA applications. CORBA-as-needed allows distributed CORBA applications to first detect if the interoperability functionality of CORBA communication methods are indeed needed in a particular invocation, and bypass them if this functionality is not needed for that invocation. The paper describes three design techniques to incorporate CORBA-as-needed in the current CORBA architecture. The paper also describes the design, implementation, and a detailed performance evaluation of a custom-made system to validate the usefulness of the CORBA-as-needed technique.

1. Introduction

CORBA has been successfully used for constructing a large number of distributed applications. However, it suffers from an important deficiency. There is a significant performance overhead in using CORBA to construct distributed applications [3, 5]. In this paper, we propose *corba-as-needed* technique to improve the performance of distributed, CORBA applications. This technique allows distributed applications to bypass CORBA communication methods whenever possible, and still preserve the interoperability of the applications being built. In particular, we propose and analyze three design techniques in the current CORBA technology that allow distributed applications to first detect if the interoperability functionality of CORBA communication methods are indeed needed in a particular invocation, and bypass them if this functionality is not needed for that invocation. After a thorough analysis of the three design techniques, we provide a detailed description of the third design technique, namely the wrapper approach.

To validate the usefulness of the wrapper approach, we have designed and implemented a middleware system that provides all necessary CORBA functionality (location transparency in particular) except interoperability. The main difference between a CORBA middleware and this custom middleware (called CM) is that CM uses TCP for communication between client and server instead of CORBA communication methods. We describe the design and implementation of CM, and then compare the performance of two versions of a representative client/server application. The first version of this application uses CORBA (Visibroker 3.4 [1]), while the second version uses CM.

Performance measured from these implementations confirm that distributed applications are expected to benefit significantly from using the proposed wrapper approach. To pinpoint the sources of performance overhead that Visibroker incurs to preserve interoperability between different computing platforms and programming languages, we performed three separate experiments. These experiments were designed to measure individually the effect of different interoperability issues in Visibroker. Results from these experiments show that parameter marshalling and unmarshalling needed for remote method invocation in Visibroker is the main source of performance overhead. We provide a detailed description of these experiments and performance analysis in the paper.

2. Motivation

While the ability to operate in a heterogeneous distributed computing environment is an important requirement for modern distributed applications, it is important to note that a majority of the applications do in fact use operating systems, network protocols, and hardware platforms that are compatible with one another. For example, a quick survey of the major servers providing services over the Internet shows that a significant majority of them use Windows operating system. In addition, judging by the commercial sale, a significant majority of the clients also use Windows operating system. The important observation here is that the interoperability property provided by CORBA is

not really needed in a majority of client-server connections, and a client-server system implemented using CORBA unnecessarily incurs performance overhead for such connections.

However, to ensure the generality of distributed applications, they must be constructed to operate in a heterogeneous, distributed computing environment. In particular, these applications must be able to operate correctly when the client and the server's computing platform are incompatible with one another. The main idea of this paper is to design a system that dynamically recognizes the situations when interoperability property of CORBA is not needed, and allows applications to simply bypass CORBA communication methods by using a standard TCP/UDP communication mechanism in those situations. In other words, interoperability property of CORBA is used by applications only when it is really needed.

3. CORBA As Needed

3.1. Design Alternatives

There are at least three different approaches to incorporate corba-as-needed technique. Figure 1 illustrates these three design approaches. In the first approach, we can implement a new CORBA service called *bypass service*. The bypass service uses a similar interface as other services in CORBA, but uses TCP or UDP as the underlying communication mechanism instead of CORBA communication methods. When a client invokes a method in the ORB, the ORB calls the naming service to locate the server and get platform information of the server. If the server is determined to be running on a compatible platform, it calls the new bypass service to facilitate the invocation. Otherwise, it uses the regular CORBA invocations. We will call this design technique the *service approach*.

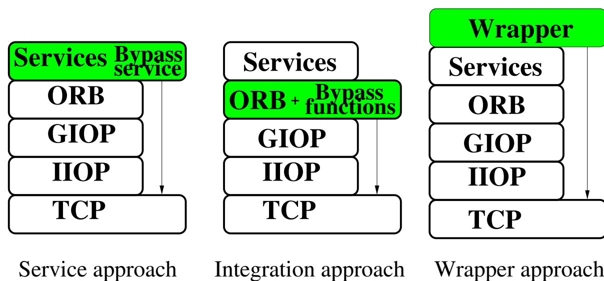

Figure 1. Design Approaches.

In the second approach, an existing ORB is modified and extended to include the functionality of determining compatibility between client and server's platforms, and facilitating all communications via TCP or UDP if client and server's platforms are determined to be compatible. We will call this design technique the *integration approach*.

Finally, in the third design approach, the functionality of determining platform compatibility, and establishing alternate connection are implemented as a new module called *wrapper*. The wrapper is interposed between a client/server and an ORB. It intercepts all method invocations between the client/server and the ORB. When a client initiates a connection with a server, the wrapper intercepts this invocation, and invokes appropriate ORB methods to determine the server location and server platform. If the server platform is compatible with the client platform, the wrapper uses TCP or UDP to facilitate all the following method invocations. We will call this design technique the *wrapper approach*.

3.2. Discussion

The integration approach is an appealing approach from performance point of view. Because all additional functionality are incorporated in the ORB itself, this approach is expected to minimize the performance overhead due to the extra functionality of determining platform compatibility, and maximize the performance benefit due to alternate TCP or UDP communication facility. However, there are several problems associated with this approach. First, this approach violates the basic philosophy behind the design of CORBA: the ORB should provide minimal functionality to allow interoperability between heterogeneous objects. By integrating additional functionality in ORB, the functionality of ORB are overloaded. A more serious problem with this approach is that it may result in the loss of portability and interoperability. Integrating additional functionality in ORB forces applications to be ORB dependent. Application clients and servers are required to use the same ORB implementation (compatible implementations of additional functionality).

The service approach avoids the portability and interoperability problem of the integration approach by not modifying an existing ORB in any significant way. It also provides for an easy integration in the CORBA service framework, and can take advantage of other CORBA services such as security. However, the main problem associated with this approach is performance overhead. In an earlier research project, we observed that providing additional functionality such as group communication as a CORBA service can result in a very significant performance overhead [4]. Another problem associated with this approach is it is not clear how ORB can interface with the Bypass service, so as to invoke that service when needed.

Finally, the wrapper service does not require any modification in an existing ORB, or creation of a new CORBA service. As a result, this approach is certainly very appealing.

The main concern is how much performance overhead will be incurred by such a wrapper service that needs to intercept every method invocation from the client. We decided to experiment with the wrapper approach, because of two reasons: (1) ease of implementation, and (2) a clean separation between the added functionality and the CORBA ORB or CORBA services.

3.3. Wrapper Approach

The wrapper approach consists of a client-side wrapper and a server-side wrapper. These two wrappers provide functions to detect if a CORBA communication method is needed for each initial invocation request, and establish an alternate TCP/UDP connection if CORBA communication method is not needed. Figure 2 illustrates the architecture of this approach. An important observation is that the client-side wrapper and the server-side wrapper make the entire process of determining if CORBA is indeed needed, establishing an alternate connection, and facilitating all future communications transparent to both the client and the server. In particular, the two wrappers perform the following functions:

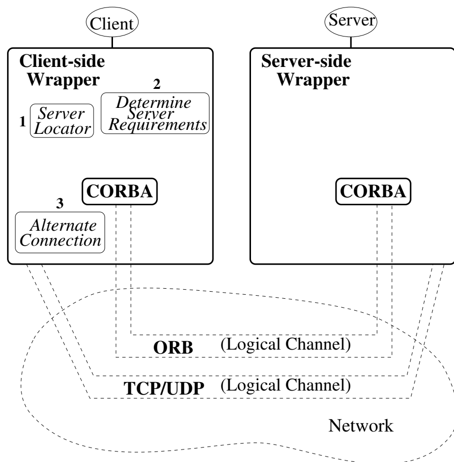

Figure 2. Wrapper Approach).

1. When a client attempts to locate a server, the client-side wrapper uses the standard CORBA services to locate the server in the network. This ensures that the CORBA location transparency is provided to all applications constructed using the wrapper.

2. When a server is found, the client wrapper determines if the client and server are running on compatible platforms.

3. If the client wrapper determines that the client and the server are not running on compatible platforms, it facilitates all future communications through CORBA. This ensures that the CORBA interoperability service is provided to all applications constructed using the wrappers, when there is some incompatibility between the client and the server platforms.

4. If the client wrapper determines that the client and server are indeed running on compatible platforms, it establishes a separate connection between the server and the client using a transport-level protocol and facilitates all future communications through this new connection. This ensures that CORBA communication methods are bypassed when the client and the server's platforms are compatible with one another.

Clearly, by introducing a wrapper, applications will have additional performance overhead. This is particularly true in the beginning when the wrapper tries to determine if heterogeneity functionality of CORBA are indeed needed. This performance overhead is later offset by the performance gains if subsequent communication does not require heterogeneity functionality of CORBA. So, the key question is how much overall performance gain will occur by using the proposed wrappers. The next two sections provide an answer to this question.

4. Custom Middleware

Our goal is to determine the performance overhead CORBA incurs to operate on a heterogeneous, distributed computing platform. To do so, we have designed and implemented a custom middleware called CM that provides support for client/server programming. Like CORBA, CM provides location transparency, but unlike CORBA, CM operates on a homogeneous, distributed computing platform. The design and implementation of CM closely follows the design and implementation of Visibroker. We have done this to ensure that the performance difference between Visibroker and CM truly reflects the performance overhead Visibroker incurs to provide the interoperability functionality.

CM uses Java socket APIs to allow Java applications to invoke Java objects over various platforms that run JVM. Like Visibroker, CM uses multiple smart agents to reliably locate servers, and thread pooling at the servers to efficiently handle multiple client requests concurrently. Figure 3 illustrates the steps involved in this invocation. These invocation steps are as follows:

1. `Client` tells `ClientWrapper` the target object name.

2. `ClientWrapper` broadcasts (UDP broadcast) a location request to all `smartAgents` in the network.

3. All `smartAgents` that have the target object address respond with the target server's address (IP address and port number).

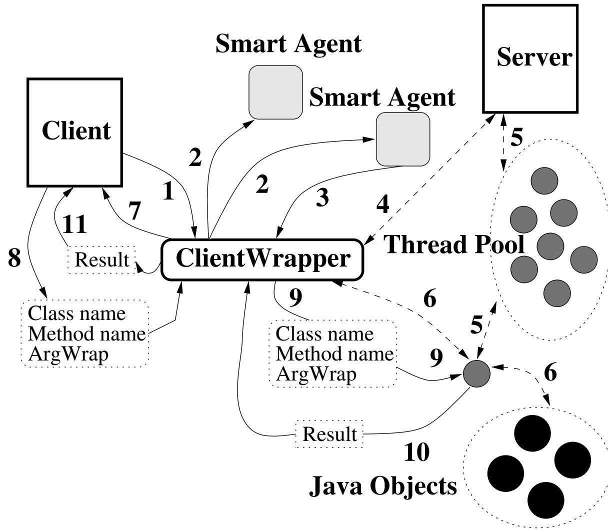

Figure 3. Invocation steps in CM.

4. `ClientWrapper` locates the target server using TCP.

5. The target server allocates a thread to establish a connection with the `ClientWrapper`.

6. The thread creates an instance of the target object and notifies the `ClientWrapper` when done.

7. `ClientWrapper` forwards this information to the `Client`.

8. `Client` tells the `ClientWrapper` the target method name to be invoked along with the necessary arguments.

9. `ClientWrapper` sends a TCP message that encapsulates the target class name, method name, and arguments to the target object.

10. The thread executes the invoked method, and sends the results back to the `ClientWrapper`.

11. `ClientWrapper` passes this result to the `Client`.

Notice that all these steps are followed in Visibroker as well. Of course, in addition to these steps Visibroker also performs some data conversions and byte stuffing to address the interoperability issue in a distributed system. A server in CM starts by creating a certain number of threads that are queued and remain idle. On receiving a client request, the server dispatches one of these threads to handle that request. CM server also uses an adaptive algorithm that is described in [2] to dynamically create and destroy threads depending on the workload. Smart agents must be created before a server can register its objects. In CM, smart agents are simple objects that maintain a list of object name to server address (IP address, port number) mappings. After creating the initial threads, a server in CM registers with one of the smart agents. Of course, the server objects must be compiled at the server side before any client requests are received. Clients using CM first initialize a Java object called `ClientWrapper`.

5. Performance Evaluation

To measure the performance difference between CM and Visibroker, we constructed an application consisting of a client program and a server program. We implemented this application using both CM and Visibroker. Both the client and the server programs are written in Java. The client program simply invokes a method `echoString`(`string` *str*) on an object `Echo` that resides on the server. When invoked, this method simply returns the string parameter *str* back to the client. The IDL interface of this method is shown in below. An object that implements this interface resides on the server.

```
Interface Echo {
  string echoString (in string str); }
```

The implementation of the client program is different for CM and Visibroker. In the implementation using Visibroker, the client simply invokes the `echoString`() method. The `Echo` object on the client side is merely an interface. The `echoString`() method dispatches its argument to a stub routine that remotely invokes the `echoString`() method on the `Echo` object that is residing on the server. In the implementation using CM, the client first creates an `ArgWrap` object and puts the `String` argument *str* in this object. It then invokes the `remoteCall`() method of the `ClientWrapper` object that invokes the `echoString`() method on the `Echo` object that is residing on the server.

Visibroker Client

```
for (int i = 0; i <50; i + +)
  String reply = Echo.echoString (str);
```

CM Client

```
ArgWrap argwrap = new ArgWrap (1);
argwrap.add ("java.lang.String", str);
for (int i = 0; i <50; i + +)
  Object reply = cWrapper.RemoteCall
      ("Echo", "echoString", argwrap);
cWrapper.KillConnection();
```

Since establishing a TCP connection consumes lot of time, and the chances of subsequent invocations on an object after the first invocation are high, Visibroker is designed

to not terminate a TCP connection until that connection remains idle for some period of time. In our experiments, we observed (via a network sniffer that is described later) that Visibroker never terminated the TCP connection. So, we designed CM to not terminate its TCP connection throughout the our experiments.

5.1. Performance

We conducted our experiments by running the client/server system with three different string lengths—100 bytes, 500 bytes, and 10K bytes, over three different networks—LAN, Cross subnet, and WAN. The LAN experiment was conducted between two adjacent workstations connected by a 10 Mbps Ethernet with in the *wks.cs.uwyo.edu* domain. Both of these workstations were running Windows NT. The cross subnet experiment was conducted between two workstations on different subnets with two routers in between. Again, both of these workstations were running Windows NT. Finally, the WAN experiment was conducted between two workstations connected by the Internet. One workstation was in Laramie, Wyoming and the other in San Francisco, California. Both of these workstations were running Linux (Redhat 6.0). All workstations used in our experiments have Intel processors that support identical natural data alignment and little-endian byte ordering.

Network	Message Size	CM	Visibroker
LAN	100 B	86.3	140.6
	500 B	93.3	177.3
	10 KB	187.0	260.0
Cross Subnet	100 B	177.0	220.3
	500 B	196.6	247.0
	10 KB	433.6	921.3
WAN	100 B	6042.3	6371.3
	500 B	6280.0	6443.6
	10 KB	6685.3	7056.6

Table 1. Application Performance (msec)

In the performance reported here, we invoked `echoString()` 50 times in a row to average out any system and network inconsistencies. Table 1 shows the performance measured from the two implementations on LAN, Cross subnet, and WAN computing environment. According to these test results, the application implementation in CM outperforms the application implementation in Visibroker by as much as 38% over LAN, 31% over cross subnet, and 5% over WAN. The key reason for this performance improvement is that Visibroker performs some data mapping, byte stuffing, and other interoperability functions to address the interoperability issue that CM does not perform. The performance improvement of CM over Visibroker reduces with increase in network latency. The main reason for this is that the time spent by Visibroker in addressing the interoperability is constant and independent of the network latency. So, as the network latency increases, the percentage of time spent on addressing the interoperability issue, and hence the overhead of Visibroker, decreases.

5.2. Evaluation

CORBA ORBs communicate with one another using the IIOP, which maps GIOP specifications onto TCP/IP. GIOP uses Common Data Representation (CDR) to manage incompatibility in data representation between different architectures and programming languages. CDR is a transfer syntax that performs mappings between OMG IDL data types and language-specific data types on different platforms. It manages three important interoperability issues: (1) variable byte ordering, (2) byte alignment of primitive types, and (3) a complete OMG IDL mapping.

To pinpoint the sources of performance overhead that Visibroker incurs to preserve interoperability between different computing platforms and programming languages, we installed a *network sniffer* on the network to examine every packet sent by Visibroker as a part of the client/server application. In addition, we performed three separate experiments to measure the effect of each of the three interoperability issues mentioned above.

We observed that irrespective of the CPU used, the byte-order entry in GIOP packets coming out of the Visibroker client or server was set to zero, i.e. the byte ordering in all packets was always big-endian. The reason for this is that the Java virtual machine (JVM) uses big-endian order internally, i.e. if an Intel processor is used (as was the case in our experiments), JVM converts byte orders from little-endian to big-endian internally. As a result, there was no extra performance overhead incurred by the application implemented using Visibroker due to variable byte orderings. To understand the effect of variable byte ordering on the performance of applications implemented using Visibroker, we experimented with another client-server application written in C++ using Visibroker. We noticed that Visibroker performs a byte ordering conversion only when the client and server CPUs support different byte orderings. So, this interoperability issue does not add any extra performance overhead on the performance of a client-server application implemented using Visibroker, if the client and server CPUs support the same byte orderings.

The second CDR specification forces data structures to be aligned based on natural primitive type boundaries and introduces byte padding if necessary. Again, after observing the GIOP packets from Visibroker client and server, we

noticed that no extra padding was introduced by Visibroker in our experiments. The reason for this is that only character strings, which do not impose any alignment restrictions, were exchanged in our experiments. To observe the effect of padding, we conducted another simple experiment that involved transferring of several complex data structures consisting of `char`, `long`, and `double` data types. This experiment showed that while byte padding for alignment does occur in Visibroker, it does not introduce any significant performance overhead. The only exception is a rare case when the extra bytes due to padding forced extra frames to be transferred over the network physical layer. So, for all practical purposes, this issue also does not add any extra performance overhead.

Finally, CDR provides a complete IDL mapping to language-specific data types. The parameter marshalling/unmarshalling becomes an issue here. During a remote invocation, the string parameter is marshalled from the `java.lang.String` type to the IDL `String` type. Similarly, on receiving the result, the IDL `String` type is unmarshalled to the `java.lang.String` type. Similar parameter marshalling/unmarshalling takes place on the server side. We noticed that the most significant performance overhead incurred by the application implemented using Visibroker is due to parameter marshalling/unmarshalling. In CM, this parameter marshalling/unmarshalling does not take place.

To explore the effect of parameter marshalling/unmarshalling in Visibroker further, we performed several simple experiments that involved marshalling/unmarshalling between several language-specific data types and IDL types. We noticed that marshalling/unmarshalling always resulted in a significant performance overhead. In particular, the IDL data type `Any` stands out here. Parameter marshalling/unmarshalling involving the `Any` data type took significant amount of time. This is because extra description needs to be transferred (in addition to the marshalling/unmarshalling overhead) when the IDL `Any` type is involved.

6. Discussion

Heterogeneity in modern, distributed computing systems is a fact of life. As a result, the ability of modern, distributed computing applications to operate in a heterogeneous, distributed computing environment is essential. CORBA is a very useful technology that enables the construction of distributed, heterogeneous applications. It naturally incurs some performance overhead to provide the heterogeneity support. This extra performance overhead has limited the applicability of CORBA. So, it is important to design techniques to address this limitation of CORBA.

So far, research in addressing this performance limitation of distributed, CORBA applications has been mainly focused on designing more efficient ORBs. In this paper, we have proposed another complementary approach that can improve the performance of distributed, CORBA applications under some common computing scenarios. The main motivation behind this approach is the observation that a majority of the components in modern, distributed computing environments are compatible with one another and can interoperate with one another. This is despite the fact that most modern, distributed computing environments are heterogeneous in general. So, while CORBA is needed to develop modern, distributed computing applications, its interoperability functionality are not really needed in a significant number of cases.

To understand the potential benefits of this approach, we have done a detailed experimental analysis of the sources of CORBA performance overhead due to the interoperability property. The main conclusion of this analysis is that the main source of performance overhead due to interoperability functionality in CORBA is parameter marshalling and unmarshalling needed in remote method invocation. The interoperability functionality results in a performance overhead of as much as 38% in a local area network, 31% in a cross-subnet network, and 5% in a wide-area network.

It is important to note that these performance overheads are quite significant. This is because a small difference in communication delays between the CORBA communication methods and UDP/TCP can result in a very significant performance overhead in higher-level systems such as group communication services implemented using CORBA. We observed in [4] that a relatively small difference in one-way communication delay between CORBA communication method of IONA OrbixORB 2.3 and UDP resulted in a 10-15 times performance degradation of the timewheel group communication service. So, even a relatively small performance improvement, such as 5% in a wide-area network, in basic CORBA communication can result in a significant reduction in the performance degradation of applications implemented using CORBA.

References

[1] *Visibroker for Java, Programmer's Guide*. Inprise Corporation, 1996.

[2] D. Bulka. *Java Performance and Scalability*, volume 1. Addison-Wesley, 2000.

[3] A. Gokhale and D. C. Schmidt. Measuring the performance of communication middleware on high-speed networks. In *Proceedings of SIGCOMM'96*, Aug 1996.

[4] S. Mishra, L. Fei, X. Lin, and G. Xing. On group communication support in CORBA. *IEEE Transaction on Parallel and Distributed Systems*, 12(2), February 2001.

[5] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. Wiley, 2nd edition, 1998.