A Communication Model for CORBA Systems

René Sáenz and Brian J. d'Auriol
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas 79968
rsaenz,bdauriol@cs.utep.edu

Abstract

A multi-component communication analysis of the CORBA specification is conducted and an analytic model describing communication costs is proposed. The analysis indicates: a) potential expensive communication when integrating parallel and distributed computing into a CORBA middleware framework, and b) a relationship between communication costs and code layout in CORBA's Interface Definition Language (IDL).

1 Introduction

In 1989 the Object Management Group (OMG), an international consortium of more than 800 members, was established to create standards for distributed systems. The OMG developed the Object Management Architecture (OMA). The OMA provides an architectural framework for object-oriented distributed systems, in particular, the OMA is the basis for the Common Object Request Broker Architecture (CORBA). CORBA defines the interaction, interfaces and characteristics of the OMA. CORBA's main components are: Object Request Broker (ORB), Interface Definition Language (IDL), Interface Repository, Implementation Repository, marshalling, stubs and skeletons, dynamic invocation, object adapter, and inter-ORB protocols. CORBA combines two main trends in the information field: object-oriented methodology and client/server computing.

There are few studies that have concentrated on assessing, quantifying or measuring the communication overhead in a CORBA system. A number of comparison projects were conducted at the Charles University at Prague Czech Republic. These projects concentrated on ORBs from different vendors under different environments. These projects mainly compare the performance of different ORBs and did not address the amount of communication overhead in a CORBA system [5, 6]. Schmidt and Gokhale have evaluated CORBA under ATM networks. In their work, they have detected inefficiencies in CORBA and have provided improvements in their development of a new ORB called The Ace ORB (TAO). Their work has mainly addressed the implementation problems in the different areas of the CORBA architecture [3, 2, 4, 1]. These studies may suffer from a lack of universality because the implementation of the ORB changes from vendor to vendor, as can be seen in [5, 6].

This paper focuses on the CORBA specification [8]. In particular, this paper reports on a communication analysis based on the CORBA specification and includes a proposed model of CORBA communications. Additional contributions include: the identification of the major communication components in a CORBA system, a classification of the communication components and its subsequent use in a further analysis of the communication overheads in CORBA. Also included are two testbench programs to assist in the analysis and application of the proposed model.

This paper is organized as follows. A brief review of CORBA is presented in Section 2. Section 3 presents our analysis of the various communication components and issues in CORBA and in addition, presents a proposed communication classification. In Section 4, an analysis of the marshalling process is conducted. In Section 5, we propose an analytic model of CORBA communication. This model is applied in Section 6 to assist in the analysis of the static and dynamic invocation mechanisms of CORBA. Conclusions are presented in Section 7.

2 Review of CORBA

Figure 1 shows an overview of the CORBA architecture. The core component of the architecture is the ORB. The figure shows how the communication of the client and server is facilitated by the ORB's different interfaces. These interfaces are:

- Stubs and Skeletons: Stubs take the requests of the client and marshal data (format typed data into packet-level representation) to the server. Skeletons on the

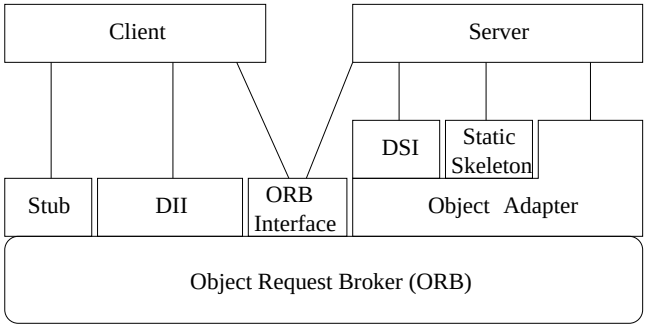

Figure 1. CORBA Architecture: Principal interfaces

server side demarshal (deformat the packet-level representation back into typed data) and pass the request to the server so that it can serviced.

- Dynamic Invocation Interface (DII): This interface is used when the client does not have knowledge of the server's interface at compile time. [8] Applications use the DII to dynamically issue requests to servers without requiring IDL interface-specific stubs and skeletons to be linked at compile time.

- Dynamic Skeleton Interface (DSI): This interface is the server's equivalent of the client's DII.

- Object Adapter: The object adapter associates server objects with the ORB. The object adapter provides the means for object reference generation, operation dispatching and activation/deactivation of server objects. The current standard object adapter is known as the Portable Object Adapter (POA).

IDL is a specification language that allows for the specification of interfaces between client and server parameters. The principal component of an IDL specification is the signature. A signature is collectively the name of the function, the return type and the data type of the parameters, if any, that the function accepts. IDL provides both basic data types as well as aggregate data types similar to those provided by many programming languages. IDL also provides for exception handling specifications.

In IDL, the flow of information between client and server is specified by the IDL keywords `in, out` and `inout` in the signatures. A parameter labeled as `in` (input) signifies that the arguments are passed from the client to the server. An `out` (output) parameter indicates that the arguments are passed from the server to the client. An `inout` (input/output) parameter supports two-way information interchange. Signatures in IDL can also use a special keyword called `oneway`. These types of operations are used for notification purposes only. Clearly, `oneway` operations cannot have in their signatures any return type values nor `out` nor `inout` parameters.

The IDL compiler/translator also may store the IDL interfaces in the interface repository. The interface repository is a service that provides for a persistent set of objects representing the IDL information.

Object references identify a CORBA object and contain addressing information (server name, machine and port numbers) on how to reach an object in a distributed system. Object references can denote either a transient or persistent CORBA object. Transient objects are those whose life cycle ends when the server shuts down. On the other hand, persistent objects are those that outlive the life cycle of the server. Persistent CORBA objects are supported by the implementation repository.

The General Inter-ORB Protocol (GIOP) describes how specific communication protocols can be mapped to fit within the GIOP framework. The mapping of GIOP to the Transmission Control Protocol/Internet Protocol (TCP/IP) is referred as the Internet Inter-ORB Protocol (IIOP). The GIOP specification consists of the following elements: GIOP transport assumptions, Common Data Representation (CDR), Message formats.

There are eight GIOP message types that are sufficient to support the interoperability between ORBs. Each GIOP message has a message header followed by a message body. The message body depends on the type of message being sent. The message header contains a special byte sequence called the magic number, GIOP version, byte order, flags, message type and message size. The eight message types are: *Request*, *LocateRequest*, *CancelRequest*, *Reply*, *LocateReply*, *CloseConnection*, *MessageError*, *Fragment*.

The GIOP protocol specifies the Common Data Representation (CDR) as its data transfer syntax. The CDR has the following main characteristics: support for byte ordering (little-endian/big-endian) representation, data alignment and complete IDL mapping.

3 CORBA Communication Analysis

The six major communication parts that we have identified in CORBA systems are shown in Figure 2 by bracketed numbers. Some of these parts are further described below.

When the POA activates a server object, it writes the object reference (transient or persistent) of the activated object to a file (IOR) (see (1) in the Figure 2). The stub marshals a request using the CDR rules When the request arrives at the server side application, the skeleton demarshals the request and delivers it to the server object (see (2) in Figure 2). In the case of persistent object references, the IOR contains an object reference that points to the machine where the imple-

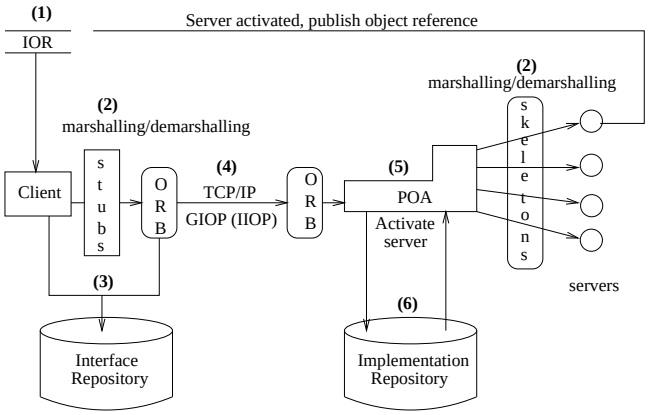

Figure 2. Identified CORBA communication components.

mentation repository is running. The implementation repository activates the server object, if needed, and sends a new IOR to the client (see (6) in Figure 2).

Figure 3 shows the proposed CORBA classification.

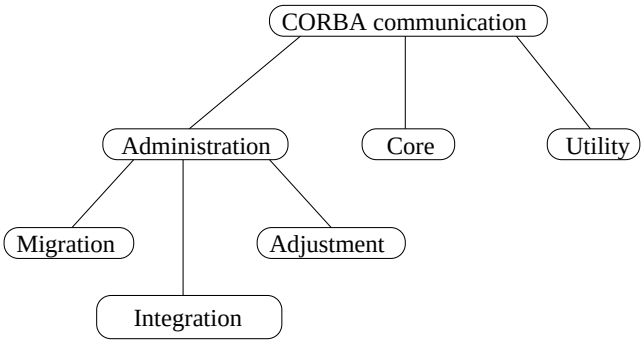

Figure 3. CORBA communication classification

The Core category includes the communication procedures, functions and/or processes that are necessary (indispensable) to have communication from client to server and back. The Utility category includes all those communication components that add flexibility to the system; it enhances the core category. The Administration category groups all the communication that supports portability, interoperability and scalability of the architecture. In other words, the Administration category describes the managerial communication parts of CORBA. The Integration subcategory classifies all the communication components necessary for one vendor ORB to communicate with another vendor ORB. The Migration subcategory classifies all the communication components necessary to support the mobility of an object from one machine to another. The Adjustment subcategory classifies all the communication components necessary for a CORBA system to be scalable.

Classification Type	Communication Component	Fig. 2 Comm. Ref.
Utility	DII, Interface Repository, DSI	(3)
	CancelRequest	(4)
	CloseConnection	(4)
	MessageError	(4)
	Fragment	(4)
Integration	IOR	(1)
	Marshalling/Demarshalling	(2)
Migration	LocateRequest	(4)
	LocateReply	(4)
	Implementation Repository	(6)
Adjustment	POA	(5)
Core	Request	(4)
	Reply	(4)

Table 1. Communication Summary

Each one of the six identified communication components in Figure 2 can be classified according to the classification in Figure 3. Table 1 presents these component classifications.

4 Empirical Marshalling Analysis

In a CORBA system, communication overhead is introduced not only by the different communication components, but also by the marshalling process. The communication performance is affected by the extra bytes that have to be transmitted due to the marshalling process. Table 2 shows the different IDL data types and their alignment in the transmit buffer; the output buffer where the marshalled message is assembled. The worst case scenario is one that has an IDL data type that is aligned at offset zero, followed by another IDL data type aligned at offset eight. This accounts for approximately 44% (9/16) of worst case padding overhead.

In Figure 4, the parameter of `op1()` is a structure that encapsulates the worst case scenario. The first member of the structure, which is a `char`, will be aligned in the first position (offset 0). The second member, which is a `long long`, will be aligned at offset eight. The spaces between offset one and offset eight need to be padded. Thus, instead of transmitting nine bytes (one for the `char` and eight for the `long long`), a total of sixteen bytes will be transmitted.

The best case scenario is one that has data types that need no padding between them. This scenario is denoted

Alignment	IDL data types
1	char, octet, boolean
2	short, unsigned short
4	long, unsigned long, float, enumerated types
8	long long, unsigned long long, double, long double
1,2 or 4	wchar (alignment depends on codeset)

Table 2. CDR alignment of primitive fixed-length types [7]

by `op2()` in Figure 4. Since `long long` data types are aligned on an eight-byte word boundary they can be aligned at offset zero, thus, no padding is needed, resulting in a 0% padding overhead.

The operation signature of `operation()`, is coded to represent the basic time required to send and receive messages with no data (GIOP messages with empty bodies).

```
module benchmark{
  struct _Struct{
   char c1;
   long long ll;
  };

  typedef sequence<_Struct> structs;
  typedef sequence<long long> longs;

  //Each operation is round trip
  interface perfBench{
    //operation call with no data
    void operation();
    //operation with an array of structs
    structs op1(in structs s);
    //operation with an array of longs
    longs op2(in longs l);
  };
};
```

Figure 4. IDL definition code for marshalling performance benchmark.

Furthermore, the order of the parameters of the operation signatures impacts marshalling. This can be shown easily in the worst case operation signature. If the order of members in the structure in `op1()` were to be reversed, the padding overhead will be zero because the `long long` data type would be aligned at offset zero and the `char` will be aligned immediately following the `long long` data type. This observation shows that the design of IDL interfaces must be considered in detail, especially when designing large systems that could transmit large amounts of data. As a result, this analysis shows that marshalling is IDL dependent.

An empirical benchmark test is designed to capture the best and worst case padding overhead scenarios in a CORBA system due to marshalling. The benchmark programs in this section were conducted using a network of 10/100 Megabits per-second (Mbps), connected to two identical UltraSPARC machines running SunOS 5.8. Each UltraSPARC contains 64 MB RAM, 9 GB HDD and a 270 MHz SPARC CPU. Also, the benchmark programs were compiled using Java 1.3.1 and ORB JacORB 1.3.29.

The padding overhead introduces a greater length message. Since both the client and server benchmark programs were executed on identical hardware and software environments, the communication transit time is expected to be proportional to the padding overhead.

The marshalling performance benchmark code of Figure 5 sends arrays of `structs` and `long long` and has the flexibility on inputing the sizes of the arrays. The lower line of Figure 5 represents the operation `operation()`. This baseline measurement represents the time require to call any method without any data. The top and middle lines of Figure 5 represent `op1()` and `op2()` respectively. It was expected to have a constant performance difference between `op1()` and `op2()` due to the padding overhead. The difference between the top and bottom lines is approximately 47%, as can be seen in Figure 5, which closely matches the 44% estimate in the worst case scenario. The entire program code for this marshalling benchmark can be found in Appendix A of [9].

During the experiment, it was noted that there is a constant stabilization "phase" before 300 operation invocations, as seen by the decreasing curve in Figure 5. Therefore, it was concluded that a range between 100 to 1000 operation calls would give reliable measurements. Also, to increase the resolution of the timing a 500 array size for the parameters was used. In each operation call, 500 `structs` and 562 `long long` parameters were transmitted. The reason for the 562 as compared to 500 is as follows. In `op1()` the total number of data bytes sent each time is 4500 (that is 9 times 500). For `op2()`, the total number of data bytes sent is 4000 (that is 8 times 500) giving a difference of 500 bytes. To provide for a comparable measurement, compensation needs to be made for the 500 structures transmitted so that the same number of data bytes is transmitted. The extra number of bytes in a structure is $500/8 = 62.5$, plus the 500 array size, for a total of 562.

5 A Communication Model

The network transmission time of marshaled messages from client to server varies depending on factors like net-

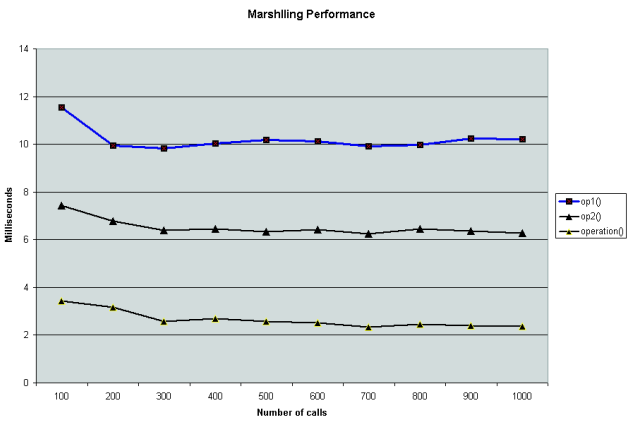

Figure 5. Testbench marshalling performance results

work speed and congestion. In this work, a constant transmission time with respect to the network layer is assumed. The transmission time is denoted by $T_{trans}(b)$ where b is the number of padded bytes due to the marshalling process.

The marshalling process that pads extra bytes when a message is CDR formatted is dependent on: the data type of the parameter, the number of parameters and the order of the parameters. The marshalling time is denoted by: $T_{marshalling}(dt, n, o)$ where dt is the data type of the parameter, n is the number of parameters and o is the order of the parameters. The marshalling/demarshalling process occurs for `in` and `out` parameters respectively. Also the `inout` parameter is marshalled/demarshalled twice. The demarshaling time is denoted by $T_{demarshalling}(dt, n, o)$

In the case of dynamic invocations, most of the processing time is done querying and/or browsing the interface repository for the construction of the request object. This time is denoted by $T_{request\_build}(dt, n)$ and is affected by the data type and number of parameters.

When a persistent CORBA object is involved in communication, added communication time due to the implementation repository must be considered. This time is denoted by T_{lookup} and represents the time it takes the implementation repository to look in an internal table for the POA name. In addition, the implementation repository adds time to execute the start-up command registered in its table. This time is denoted by T_{exec} and is further broken down into: $T_{HostConn}$, $T_{HostExec}$ and $T_{ServerUp}$. $T_{HostConn}$ denotes the time it takes the implementation repository to establish a connection to the remote host. $T_{HostExec}$ denotes the time to execute the start-up command and start the server at the remote host. T_{HostUp} denotes the time it takes the implementation repository to receive the updated information from the remote host and update its internal table.

Furthermore, the interaction between the client and the implementation repository can occur more than once. Therefore the time of the implementation repository is given by: $\sum_{i=1}^{m}(T_{ImpRep_i})$ where T_{ImpRep} is $T_{lookup} + T_{exec}$

The POA communication time is described by

$$T_{poa}(p_1, p_2, p_3, p_4, p_5, p_6, p_7) + T_{man} \quad (1)$$

where T_{man} is the time the POA manager takes to route requests to a POA and p_1 through p_7 represent the POA policies.

There are four major characteristics that impact the communication time in a CORBA system, these are: Static invocation, Dynamic invocation, Persistent CORBA objects, and Transient CORBA objects. In the following, `TRAN` refers to transient and `PER` refers to persistent.

Static Invocation and Transient CORBA Object

$$\begin{aligned} T_{total} &= T_{marshalling}(dt, n, o) + T_{trans}(b) + \\ &\quad T_{demarshalling}(dt, o, n) + T_{man} + \\ &\quad T_{poa}(TRAN, p_2, p_3, p_4, p_5, p_6, p_7) \end{aligned} \quad (2)$$

Static Invocation and Persistent CORBA Object

$$\begin{aligned} T_{total} &= T_{marshalling}(dt, n, o) + T_{trans}(b) + \\ &\quad T_{demarshalling}(dt, o, n) + T_{man} + \\ &\quad T_{poa}(PER, p_2, p_3, p_4, p_5, p_6, p_7) + \\ &\quad \sum_{i=1}^{m}(T_{lookup_i} + T_{HostConn_i} + \\ &\qquad T_{HostExec_i} + T_{ServerUp_i}) \end{aligned} \quad (3)$$

The summation in Equation 3 is due to the implementation repository and captures the number of times the client connects to different implementation repositories before the reference to a server object is bound.

Dynamic Invocation and Transient CORBA object

$$\begin{aligned} T_{total} &= T_{request\_build}(dt, n) + \\ &\quad T_{marshalling}(dt, n, o) + T_{trans}(b) + \\ &\quad T_{demarshalling}(dt, o, n) + T_{man} + \\ &\quad T_{poa}(TRAN, p_2, p_3, p_4, p_5, p_6, p_7) \end{aligned} \quad (4)$$

Dynamic Invocation and Persistent CORBA object

$$\begin{aligned} T_{total} &= T_{request\_build}(dt, n) + \\ &\quad T_{marshalling}(dt, n, o) + T_{trans}(b) + \\ &\quad T_{demarshalling}(dt, o, n) + T_{man} + \end{aligned}$$

$$T_{poa}(PER, p_2, p_3, p_4, p_5, p_6, p_7) + \sum_{i=1}^{m}(T_{lookup_i} + T_{HostConn_i} + T_{HostExec_i} + T_{ServerUp_i}) \quad (5)$$

We consider these equations to capture four broad areas of CORBA communication, that is, the transmission times, dynamic invocation times, POA and related management times, and times due to object persistence. With the exception of the IOR, these equations incorporate the communication analysis presented in Table 1 (The IOR was not considered at this point in time due to its possible out-of-the-CORBA-framework communication, see Section 3).

6 Static and Dynamic Invocation Performance Testbench

The purpose of this benchmark is to quantify the performance difference between the static and dynamic operation invocations. Our efforts here are motivated by considering the differences between Equations 2 and 4. In particular, we wish to quantify $T_{request\_build}(dt, n)$

Figure 6 contains the IDL code for the second benchmark program, containing the operation `operation()`. This operation takes a `double` as a parameter. A `double` IDL data type is also aligned in an eight-byte word boundary, capturing the same padding behavior as a `long long` would.

```
module benchtest{
   interface performance{
      double operation(in double d);
   };
};
```

Figure 6. IDL definition code for static/dynamic operation invocations.

Figure 7 shows the implementation of the operations for the performance analysis of static and dynamic invocations.

The entire program code of the marshalling benchmark may be found in Appendix B of [9]. The operation defined in Figure 6 was invoked from 100 to 1000 times (to be consistent with the first benchmark) first by static and then by dynamic invocation. The system and software used is the same as that for the earlier testbench, see Section 4.

Figure 8 shows the results of the execution of the invocation benchmark. The top line represents the dynamic invocation time and the lower line represents the static invocation time. The measurement times of the two invocation operation methods (static/dynamic) were averaged and

```
package benchtest;
public class performanceImpl
             extends performancePOA{
  public double operation(double d){
    return d;
  }
}
```

Figure 7. Implementation code for static/dynamic operation invocations.

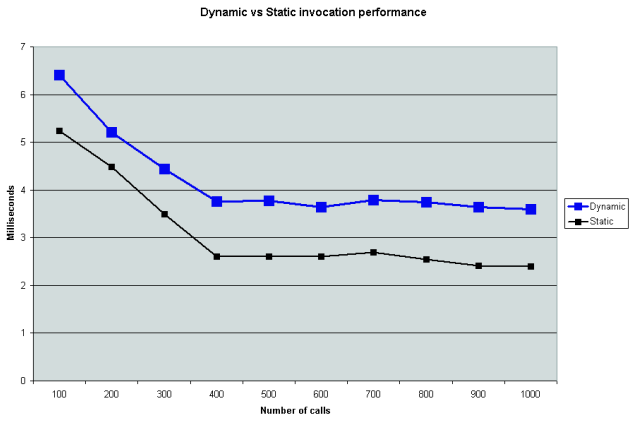

Figure 8. Static/Dynamic performance results

plotted in a similar manner as in benchmark program one. There is a 33% difference between the top and lower lines. This result can be seen in Figure 8. The design of this experiment provides for 0% padding overhead, so the 33% indicates the overhead when using the DII and DSI mechanisms to invoke an operation. Further overhead when using the dynamic invocation interfaces can be induced when highly structured data types are handled (not part of the experiment). As in the earlier case, a stabilization phase is also noted. The experiment was conducted in a similar fashion as before, that is, to iterate from 100 to 1000 operation invocations.

7 Conclusion

This paper investigates the communication overhead in CORBA systems based on a review of the CORBA specification [8]. This type of analysis differs from other related work in that other work primarily concentrates on architectural implementation factors. In this paper a multi-component communication analysis consisting of: a) an analysis of communication requirements, b) a functional

based classification model and c) analytic model describing communication costs is proposed.

Table 1 shows that out of the thirteen communication components, only two are classified as core, giving approximately 85% of the communication components classified as non-core. Table 1 also shows the categories that provide for the most non-core communication components. These categories are the Utility and Migration categories, accounting for approximately 38% and 23% respectively of the total communication components.

One implication from the analytic model is that marshalling and demarshalling may have a significant impact on communication transmission times and furthermore, such marshalling and demarshalling is dependent upon the IDL specification. A testbench program discussed in this paper provided empirical confirmation of the 44% worst case impact as well as the IDL related marshalling dependency. Additionally, this model indicates that communication costs in a CORBA system are derived from the interactions of a number of distinct system components. A second testbench program provided a quantification of the overhead of one specific system component, that being dynamic interface. In many cases, these components provide for extra features beyond the necessity of strict client/server required communications.

This analysis shows that CORBA might be very expensive in terms of communication especially in the situations where the communication is significantly composed of non-core components. Such potentially expensive communication costs may impact the ability of integrating parallel and distributed computing into a CORBA middleware framework. More work would need to be done in this area before stronger characterizations can be stated. Future work extending from this paper could include additional measurements of the model applied to two different ORBs using one for providing parameter quantification and the second, for model verification.

References

[1] A. Gokhale and D. Schmidt. The performance of the corba dynamic invocation interface and dynamic skeleton interface over high-speed atm networks. Nov 1996.

[2] A. Gokhale and D. Schmidt. Evaluating corba latency and scalability over high-speed ATM networks. In *ICDCS*, Baltimore, Maryland, USA, May 1997.

[3] A. Gokhale and D. Schmidt. Measuring and optimizing corba latency and scalability over high-speed networks. *IEEE Transactions on Computers*, 47(4), April 1998.

[4] A. Gokhale and D. Schmidt. Optomizing a corba internet inter-orb protocol (iiop) engine for minimal footprint embedded multimedia systems. *IEEE Journal on Selected Areas in Communications*, 17(9), September 1999.

[5] D. S. R. Group. Corba comparison project. Technical report, Malostranske Namesti 25, June 1998.

[6] D. S. R. Group. Corba comparison project. Technical report, Malostranske Namesti 25, July-August 1999.

[7] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*, volume 1 of *Professional Computational Series*. Addision-Wesley, 1 edition, April 1999.

[8] OMG. *The Common Object Request Broker Architecture and Specification*, 1999. http://www.omg.org.

[9] R. Sáenz. A communication analysis of corba systems. Master's thesis, University of Texas at El Paso, 2001.