The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks

Aniruddha Gokhale and Douglas C. Schmidt

gokhale@cs.wustl.edu and schmidt@cs.wustl.edu
Department of Computer Science, Box 1045
Washington University
St. Louis, MO 63130-4899, USA
Phone: (314) 935-6160 Fax: (314) 935-7302

An subset of this paper appeared in the GLOBECOM Conference, London, November $18 - 22^{nd}$ 1996.

Abstract

The Common Object Request Broker Architecture (CORBA) is intended to simplify the task of developing distributed applications. Although it is well-suited for conventional RPC-style applications, several limitations become evident when CORBA is used for a broader range of performance-sensitive applications running in heterogeneous environments over high-speed networks. This paper illustrates the performance limitations of existing CORBA implementations in terms of their support for the dynamic invocation interface and the dynamic skeleton interface. The results described below indicate that ORB implementors must optimize both the DII and DSI significantly before CORBA will be suitable for performance-sensitive applications on high-speed networks. In addition, the CORBA 2.0 DII specification must be clarified in order to ensure application portability and optimal performance.

1 Introduction

The Common Object Request Broker Architecture (CORBA) [9]) is a promising approach for improving the flexibility, reliability, and portability of communication software. CORBA is designed as an open standard for distributed object computing. It automates many common network programming tasks such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshalling and demarshalling; and operation dispatching.

Our experience over the past several years [13, 12] indicates that CORBA is well-suited for statically typed, synchronous request/response applications that run over conventional low-speed networks. However, as developers increasingly attempt to use CORBA for more dynamic types of distributed applications that run over high-speed networks several performance limitations have become evident. This paper quantifies the performance of two widely used existing CORBA implementations (Orbix 2.0 and ORBeline 2.0) in terms of their support for the *dynamic invocation interface* and the *dynamic skeleton interface*, which are described below.

• **The Dynamic Invocation Interface:** CORBA provides two different interfaces for clients to communicate with servers:

- *Static Invocation Interface (SII)* – is provided by static stubs generated by a CORBA IDL compiler and is useful when client applications know the interface offered by the server at compile-time.
- *Dynamic Invocation Interface (DII)* – is provided by an ORB's dynamic messaging mechanism and is useful when client applications do not have compile-time knowledge of the interfaces offered by servers.

Many distributed applications can be written using CORBA's SII. However, there is an important class of applications (such as network management MIB browsers, configuration management tools and distributed visualization tools and debuggers) that are easier to develop using CORBA's DII. The DII enables applications to construct and invoke CORBA requests at run-time by querying the *Interface Repository* of a server.

In addition, the DII is required for applications that use CORBA's *deferred synchronous* model of operation invocation. Although all operation invocations in CORBA are synchronous, there are three models of invocation:

- *Twoway synchronous* – which is the typical request/response model associated with RPC toolkits, where the client blocks after sending the request until the response arrives from the server;
- *Oneway synchronous* – which is a "request-only" messaging model where the client does not receive a response;
- *Deferred synchronous* – this model of twoway call decouples the request from the response so that other client processing can occur until the server sends the response.

The DII supports all three forms of invocation semantics, whereas the SII stubs only support twoway and oneway synchronous calls. Note that neither oneway nor deferred synchronous CORBA calls are asynchronous since the CORBA specification permits the ORB to block while sending a oneway call (*e.g.,* if the underlying transport layer connection is flow controlled).

The first set of experiments in this paper pinpoint precisely where the key sources of overhead exist in CORBA using the DII. These experiments reveal that the CORBA implementations spent considerable amounts of time in presentation layer conversions for various data types. The experimental results indicate that throughput for different data types is significantly different. This is due to the differences in the amount of overhead in presentation conversion for different data types.

• **The Dynamic Skeleton Interface:** A dynamic skeleton interface (DSI) is the server equivalent of the client side's dynamic invocation interface. The DSI is used when an object implementation does not have any compile time knowledge of the type of the object it is implementing. In this case, the DSI provides a mechanism for delivering incoming requests from the ORB to the object. A client in this case has no way to determine if the server was using a type-specific skeleton or a DSI.

The second set of experiments in this paper pinpoint precisely where the key sources of overhead exist in CORBA for a client using DII and a server using DSI using twoway requests. These experiments reveal that the CORBA implementations spent considerable amounts of time in presentation layer conversions for various data types. In addition, non-optimal internal buffer management strategies results in large amount of time spent for network writes and reads.

Our results indicate that the Orbix and ORBeline CORBA implementations incur very different amounts of overhead depending on the interpretation of the CORBA 2.0 DII/DSI specification and type of the data exchanged between client and server. For instance, CORBA 2.0 does not specify if a DII request object can be reused after an operation is invoked on it. ORBeline allows reuse of DII requests, whereas Orbix does not. This design decision has a substantial impact on performance, *e.g.,* ORBeline with DII request reuse consistently achieved twice as much throughput as the Orbix version that did not reuse DII requests. In contrast, the throughput for sending `shorts` in Orbix for a buffer size of 128 K is 34 Mbps as opposed to 7 Mbps in the equivalent[1] ORBeline version. In addition, both implementations of CORBA performed very poorly (between 1 to 4 Mbps) when sending `structs`. Finally, twoway request transfers achieved around 80% of the oneway throughput for each data type, due primarily to the impact of ORB latency.

The remainder of this paper is organized as follows: Section 2 outlines the key components in the CORBA communication architecture; Section 3 demonstrates the key sources of overhead in conventional CORBA implementations over ATM; Section 4 describes related work; and Section 5 presents concluding remarks.

[1] We performed two sets of DII experiments for ORBeline: one that reused DII requests and one that did not – the latter is equivalent to the Orbix behavior.

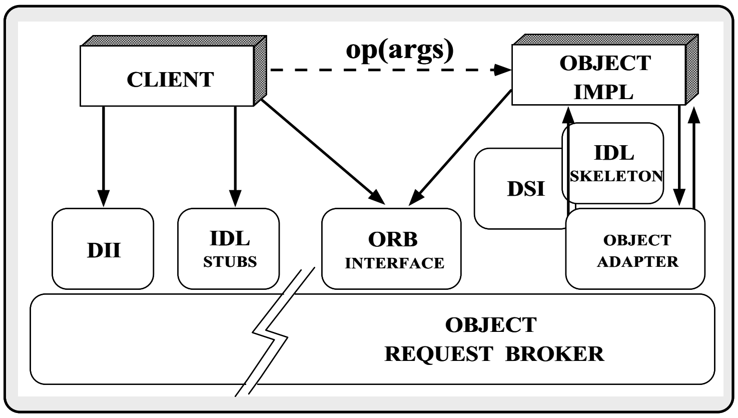

Figure 1: Components in the CORBA Distributed Object Computing Model

2 Overview of CORBA

The CORBA 2.0 standard [9] defines a set of components that allow client applications to invoke operations (`op`) with arguments (`args`) on object implementations. Object implementations can be configured to run locally and/or remotely without affecting their implementation or use. Figure 1 illustrates the primary components in the CORBA architecture. The responsibility of each component in CORBA is described below:

• **Object Implementation:** This defines operations that implement a CORBA IDL interface. Object implementations can be written in a variety of languages including C, C++, Java, Smalltalk, and Ada.

• **Client:** This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Ideally, it should be as simple as calling a method on an object, *i.e.,* `obj->op(args)`. The remaining components in Figure 1 help to support this level of transparency.

• **Object Request Broker (ORB):** When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.

• **ORB Interface:** An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB that provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

• **CORBA IDL stubs and skeletons:** Many applications utilize the IDL stubs and skeletons in CORBA's Static Invocation Interface (SII). These SII components serve as the "glue" between the client and server applications, respectively, and the ORB. The transformation between CORBA IDL definitions and the target programming language is automated by a compiler for CORBA IDL.

• **Dynamic Invocation Interface (DII):** This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike the SII IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking *deferred synchronous* (separate send and receive) operations.

• **Dynamic Skeleton Interface (DSI):** this is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.

• **Object Adapter:** associates object implementations with the ORB and assists the ORB with delivering requests to the object and with activating the object.

As shown below, the primary drawback to using CORBA is its relatively poor performance over high-speed networks. In particular, CORBA implementations of the DII and DSI perform substantially worse than the SII [5]. Moreover, the SII itself does not perform well relative to programming with lower-level communication mechanisms (such as using sockets directly over TCP/IP).

3 Experimental Results of CORBA over ATM

3.1 CORBA/ATM Testbed

This section describes our CORBA/ATM testbed and presents the results of our performance experiments. The experiments in this section were collected using a Bay Networks LattisCell 10114 ATM switch connected to two dual-processor SPARCstation 20 Model 712s running SunOS 5.5. The LattisCell 10114 is a 16 Port, OC3 155Mbs/port switch. Each SPARCstation 20 contains two 70 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.5 TCP/IP protocol stack is implemented using the STREAMS communication framework. Each SPARCstation has 128 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card.

To approximate the performance of communication middleware for channel speeds greater than our available ATM network, we also duplicated our experiments in a `loopback` mode using the I/O backplane of a dual-CPU SPARCstation 20s as a high-speed "network." The user-level memory-to-memory bandwidth of our SPARCstation 20 model 712s were measured at 1.4 Gbps, which is comparable to OC24 gigabit ATM networks [11].

3.2 Traffic Generators

Earlier studies [13, 12] tested the performance of transferring untyped bytestream data between hosts using several implementations of CORBA and other lower-level mechanisms like sockets. However, there were several limitations with these experiments:

- *Lack of presentation layer measurements* – typed data reveals the overhead imposed by the presentation layer conversions since it must be marshalled and demarshalled. However, earlier studies only examined untyped data.
- *Lack of DII measurements* – Earlier studies used the CORBA IDL compiler generated SII stubs that are unable to test important CORBA DII based applications such as network management and distributed visualization/debugging tools.
- *Lack of DSI measurements* – Earlier studies used the CORBA IDL compiler generated skeletons for the server side. No measurements were performed to evaluate the performance of the DSI.

In contrast, the experiments described below used richly typed data to identify overheads imposed by the DII and DSI. The experiments conducted for this paper extend earlier studies by measuring the performance of two different implementations of CORBA (Orbix 2.0 and ORBeline 2.0) for two features of CORBA:

• **The Dynamic Invocation Interface:** Both CORBA implementations of the CORBA-TTCP [12][2] client send 64 MB of richly-typed and untyped data as a `sequence` data type using the DII. The client tests the `invoke` (twoway communication) and `send_oneway` (oneway communication) methods supported by the CORBA `Request` interface. The server uses the skeletons generated by the IDL compiler. To use the DII, a client must first create a request using the CORBA `create_request` method. This request is then populated by the appropriate method name and its parameters in the correct order. In our experiment, we create a request and populate it with the method name and a data buffer of the desired size. We sent the same request repeatedly, only varying the data buffer, until 64 MB of data is sent to the server.

Since the request object (*i.e.,* the operation name and associated internal state) does not change in our tests, it is

[2]CORBA-TTCP is a freely available benchmarking program we developed to evaluate the performance of CORBA implementations over TCP/IP/ATM networks.

desirable to reuse the request object on every invocation to amortize the cost of creating the object. However, the CORBA specification [9] does not specify whether a request created using the `create_request` method can be reused or whether it must be created new and populated by the parameters, and released after every invocation.

Orbix 2.0 did not allow reuse of the request. Therefore, we had to create a new request and populate it repeatedly. This had an adverse effect on the performance since the client spends a significant amount of time in the request creation, population, and release. Conversely, ORBeline 2.0 did allow reuse of the request object, so we could just replace the data buffer. We performed two sets of experiments using the ORBeline CORBA-TTCP client - both with and without request reuse. As described in Section 3.4.1, ORBeline with request reuse twice as well compared with the versions not reusing requests.

• **The Dynamic Skeleton Interface:** The same ORBeline[3] client in the DII experiment was used to send requests to the ORBeline CORBA-TTCP server, which uses DSI. The DII client tests the `invoke` (twoway communication) method[4] supported by the CORBA `Request` interface. We reused the requests in these tests, as described above.

Traffic for the experiments was generated and consumed by an extended version of the CORBA-TTCP [12] protocol benchmarking tool. This tool extended the standard TTCP for use with CORBA implementations like Orbix 2.0 and ORBeline 2.0. CORBA-TTCP measures end-to-end data transfer throughput in Mbps from a transmitter process to a remote receiver process across an ATM network. In the oneway case, the flow of user data for each version of CORBA-TTCP is uni-directional, with the transmitter flooding the receiver with a user-specified number of data buffers. The twoway communication is just like the oneway communication except that additionally the client blocks until it receives an acknowledgement from the server. Various sender and receiver parameters may be selected at run-time which include the size of the socket transmit and receive queues, the number of data buffers transmitted, the size of data buffers, and the type of data in the buffers.

The following data types were used for all the tests: scalars (`short`, `char`, `long`, `octet`, `double`) and a `struct` composed of all the scalars (`PerfStruct`). The CORBA implementation transferred the data types using IDL `sequences`, which are dynamically-sized arrays. These definitions are shown in the Appendix.

3.3 Parameter Settings

Existing studies [4, 7, 2, 13, 12] of transport protocol performance over ATM demonstrate the impact of parameters such as socket queue sizes and data buffer on performance. Therefore, our CORBA-TTCP benchmarks varied these two parameters for each type of data as follows:

• **Socket queue size:** the sender and receiver socket queue size used was 64 K bytes, which is the maximum on SunOS 5.5. These parameters are shown to have a significant impact on CORBA-level and TCP-level performance on high-speed networks due to their influence on the size of the TCP segment window [7, 13].

• **Data buffer size:** Sender buffers were incremented by powers of two, ranging from 1 K bytes to 128 K bytes. The experiment was carried out ten times for each buffer size to account for variations in ATM network traffic (which was insignificant since the network was otherwise unused). The average of the throughput results is reported in the figures below.

3.4 Performance Results

The throughput measurements for the DII and DSI features of CORBA using remote transfers are presented in this section. Detailed profiling measurements of presentation layer, data copying, and memory management overhead are also presented. The profile information was obtained using the `Quantify` performance measurement tool [14]. `Quantify` analyzes performance bottlenecks and identifies sections of code that dominate execution time without including its overhead as is the case with traditional sampling-based profilers like the UNIX `gprof` tool.

3.4.1 Dynamic Invocation Interface Measurements

The figures presented in this section depict the observed user-level throughput at the sender for buffer sizes of 1 K, 2 K, 4 K, 8 K, 16 K, 32 K, 64 K and 128 K bytes using 64 KB sender and receiver socket queues (the maximum possible on SunOS 5.5).[5]

Figures 2 and 3 and 6 depict the throughput obtained for sending 64 MB of data of different data types for the Orbix and ORBeline implementations of CORBA-TTCP over ATM using oneway[6] and twoway communication without `request` reuse, respectively. Figure 4 depicts the throughput for the ORBeline CORBA-TTCP implementation that reuses the request.

The shape of the CORBA DII performance curves indicate that the observed throughputs differ significantly for different types of data. The observed throughput for a given data type depends on how the CORBA implementation performs

[3] Orbix 2.0 does not yet support DSI, so we could not perform the DSI tests for Orbix.

[4] We observed many requests were not reaching the ORBeline DSI server since the requests were `oneway`. Hence we did not report results for this tests.

[5] Our tests revealed that the receiver-side throughput was approximately the same as the sender-side. Therefore, we only show sender-side throughput results.

[6] The twoway results are consistently lower than the oneway results since the client in the twoway case blocks for an acknowledgement. Therefore, we omitted the results due to lack of space. Complete results for the tests (*i.e.*, twoway results for DII, loopback tests for the oneway and twoway DII, and oneway results for DSI) are available at `http://www.wustl.edu/~schmidt/GLOBECOM-96.ps.gz`.

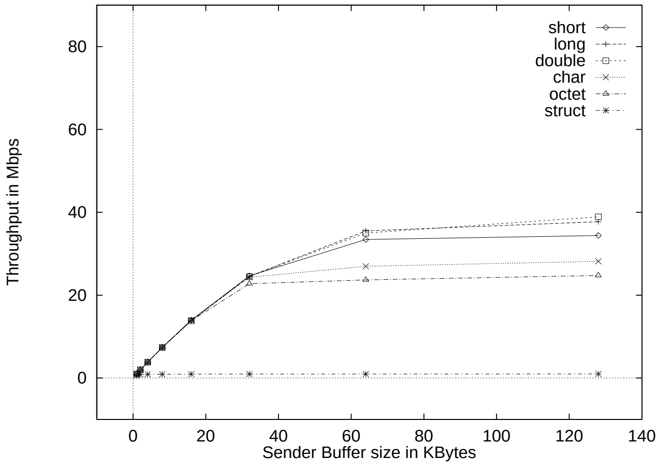

Figure 2: Orbix: Client-side Throughput for Oneway Communication using DII without Request Reuse.

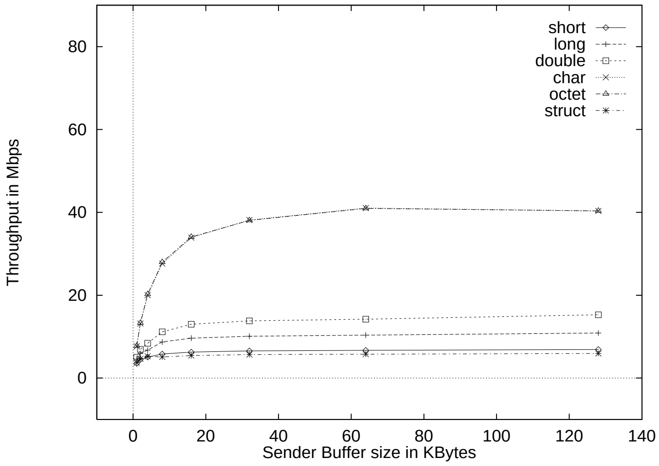

Figure 3: ORBeline: Client-side Throughput for Oneway Communication using DII without Request Reuse.

presentation layer conversions, data copying and memory management. The throughputs initially increase as the sender buffer size increases due to a decrease in the number of `writes`. However, this throughput increase begins to level off once network-layer fragmentation occurs. For the ATM network we used, the maximum transfer unit (MTU) was 9,180 bytes. Thus, fragmentation starts once the write buffers become greater than the MTU size. Although the MTU is 9,180 bytes, we do not observe the highest throughput for sender buffer sizes of 8 K, since for these buffer sizes, the number of `writes` is large and the presentation layer overhead plays a major role in decreasing the throughput.

The throughput for the oneway transfer case was consistently higher than that of the twoway case. In the oneway transfer, the client does not block for a response from the server after it has sent a request. This allows the client to send the next request immediately. This behavior differs from our CORBA twoway communication tests, where the clients are blocked until they receive an acknowledgement from the server (although no return values are sent back). The additional latency associated with this blocking reduced the throughput by roughly 20% for the twoway case. This latency may be hard to avoid, however, if the client requires confirmation that the server has completed the request.

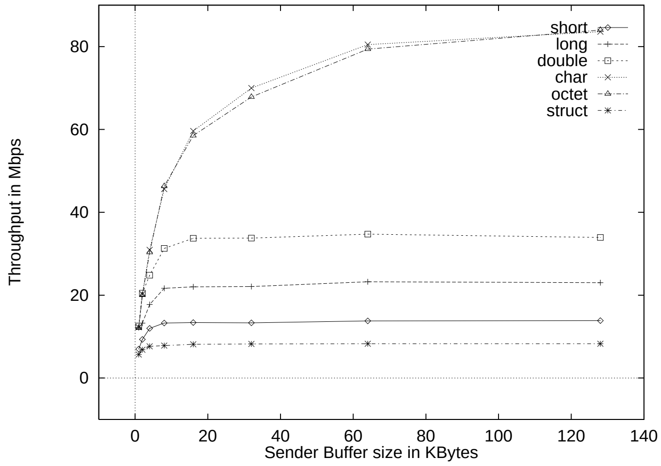

Figure 4: ORBeline: Client-side Throughput for Oneway Communication using DII with Request Reuse.

A detailed analysis of the overhead incurred on the client and the server side in sending the `octet` and `struct` data types for the oneway case with 128 K sender buffer sizes is presented in Tables 1 and 2, respectively. [7] In Tables 1 and 2, the `Time in msec` column indicates the total time spent by the corresponding operation indicated under the `category` column. The `Percent execution` column indicates the percentage of the total execution time taken up by the method. Table 1 reveals that for sending `octet/char`, the Orbix and both versions of ORBeline spent most of their time in `write` and populating the request.

In contrast, for each of the other data types (only `structs` are shown in the table), the implementations spent a significant amount of time creating a request, populating it with parameters and marshalling the data. For example, to send 64 MB of `structs`, the Orbix version spent 96% of its time in marshalling its parameter and packetizing the data. The throughput for the ORBeline version without request reuse is comparable to that of Orbix. The ORBeline version with request reuse performs much better than the ORBeline version without request reuse and the Orbix version. This is due to the less amount of time spent in marshalling (14,168 msec) and memcpy (896 msec) as opposed to the time taken by the ORBeline version that does not reuse requests (22,215 msec for marshalling and 13,428 msec for `memcpy`).

In addition, the tables reveal that the throughput for `shorts` is lower than that for `longs` because the CORBA

[7] The analysis for the twoway case was similar to the oneway case.

Version	Data Type	Analysis Category	Time (msec)	Percent Execution
Orbix	octet	write	26,728	73.52
		Populating and marshalling	7,160	19.70
		Allocate memory	1,164	3.15
		memcpy	895	2.46
	struct	Marshalling and packetizing	397,724	96.6
		write	5,016	1.23
ORBeline (without request reuse)	octet	writev	4,623	61.14
		memcpy	1,791	23.69
		Allocate memory	940	12.43
	struct	writev	77,605	55.72
		Populating and marshalling	22,215	15.95
		memcpy	13,428	9.64
ORBeline (with request reuse)	octet	writev	5,794	85.83
		memcpy	898	13.31
	struct	writev	76,575	64.89
		Populating and marshalling	14,168	12.01

Table 1: Analysis of Client-side Overheads for CORBA DII

Version	Data Type	Analysis Category	Time (msec)	Percent Execution
Orbix	octet	typecode assertion	3,361	50.00
		read	2,140	32.00
		memcpy	896	13.39
	struct	demarshalling	11,849	49.00
		typecode assertion	6,665	27.57
		read	4,985	20.62
ORBeline (without request reuse)	octet	read	3,177	90.48
	struct	demarshalling	10,552	58.52
		memcpy	3,878	21.5
		read	3,318	18.4
ORBeline (with request reuse)	octet	read	3,238	90.60
	struct	demarshalling	10,552	58.52
		memcpy	3,878	21.41
		read	3,395	18.74

Table 2: Analysis of Server-side Overheads for CORBA DII

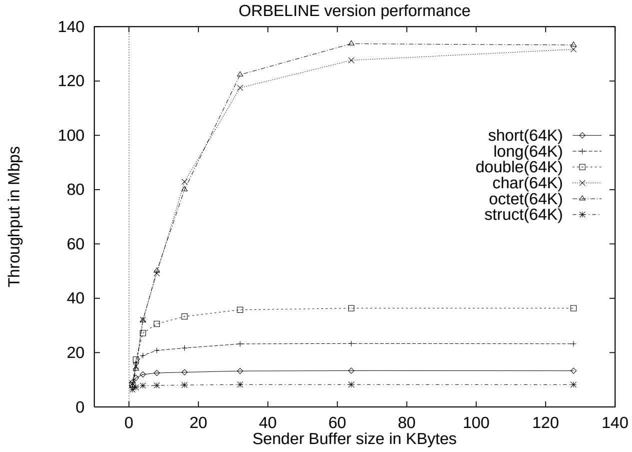

Figure 5: Client-side Throughput for Oneway Communication using DII in Loopback Mode.

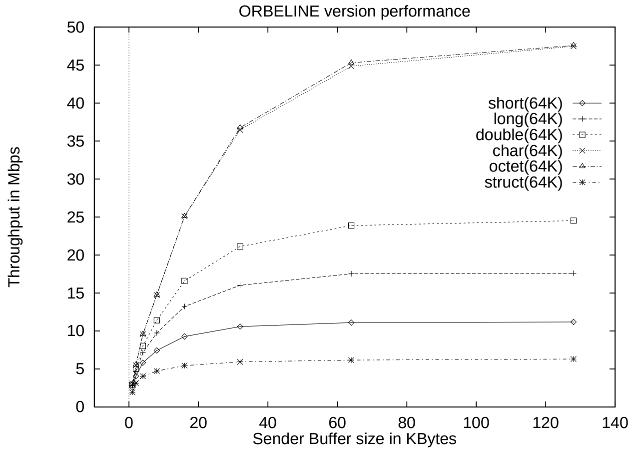

Figure 6: Client-side Throughput for Twoway Communication using DII.

DII implementation spent a larger amount of time doing marshalling and data copying for `shorts` than for `longs`. The lowest throughputs were observed for sending `structs`. The analysis of overheads for the server-side is similar to that of the client-side. The receiver side for data types other than `char/octet` spends a significant amount of time performing `reads` and demarshalling the data.

• **Loopback Results:** Figures 5 and 7 depict the throughput obtained for sending 64MB data of various data types for the ORBeline implementation[8] of CORBA-TTCP over ATM. The loopback results provide an insight into the performance that can be expected of the CORBA DII implementations for network channel speeds greater than the 155 Mbps ATM network available for our research.

The figures indicate that for sending `octet/chars`, the observed throughput increases as the sender buffer sizes in-

[8]As noted earlier, Orbix DII did not operate correctly.

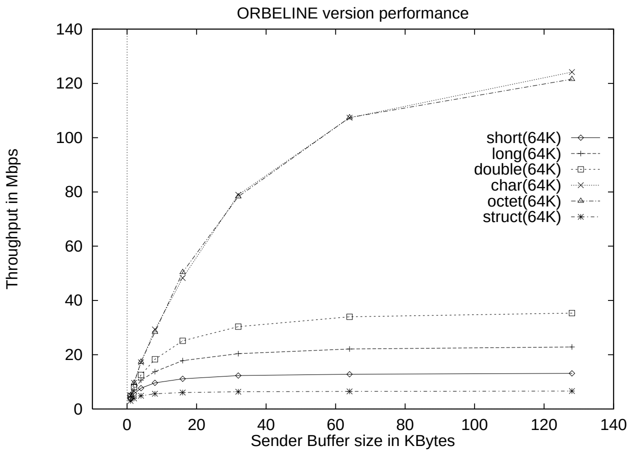

Figure 7: Client-side Throughput for Twoway Communication using DII in Loopback Mode.

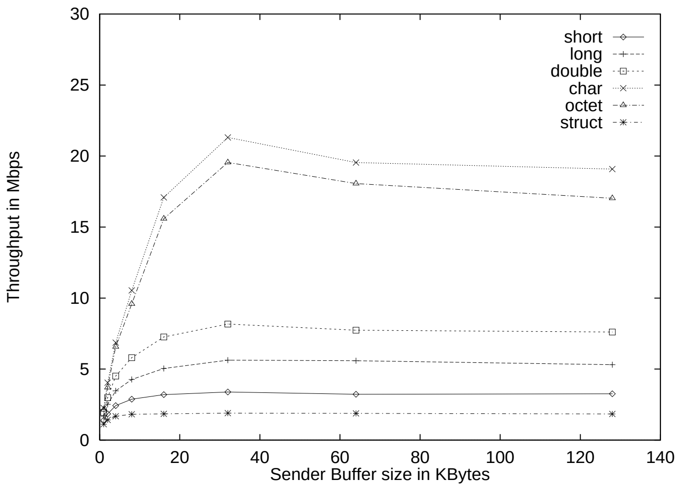

Figure 8: Observed Client Throughput using ORBeline DSI

crease. This is due to the lesser number of writes needed as sender buffer size increases. Also, due to the loopback mode, the maximum transfer unit (MTU) of the ATM network (9,180 bytes) does not cause any fragmentation and hence the throughput improves with increasing buffer sizes. The figures also indicate that for the rest of the data types, the presentation layer conversions and data copying operations severely restrict the throughput. A comparison between the `Quantify` analysis for sending `char` and `double` for 64 K sender buffer size reveals that `chars` spend 1,207 msec in `memcpy` as opposed to 7,208 msec taken up by `doubles`. In addition, the marshalling of `chars` take up 0.54 msec as opposed to 4,434 msec for `doubles`. This explains why the throughput for sending `char/octet` is much higher than that for the rest of the data types.

3.4.2 Dynamic Skeleton Interface Measurements:

Figure 8 depicts the client-side throughput for an ORBeline client using DII and an ORBeline server using DSI for twoway transfer.[9] The shape of the CORBA DSI performance curves indicate that the observed throughputs differ significantly for different types of data. The throughput for a given data type depends on how the CORBA implementation performs presentation layer conversions, data copying and memory management.

Agent	Data Type	Analysis		
		Category	Time (msec)	Percent Execution
Client	octet	read	3,911	47.95
		writev	2,923	35.84
		memcpy	912	11.19
	struct	writev	505,914	91.34
		marshalling and populating	13,341	2.41
		memcpy	7,198	1.30
		read	5,062	0.91
Server	octet	writev	23,572	82.00
		memcpy	2,701	9.40
		read	1,806	6.29
	struct	writev	1,250,130	92.41
		demarshalling	39,373	2.89
		memcpy	19,693	1.46

Table 3: Analysis of Overhead using Twoway Client-side DII and Server-Side DSI

Of all the CORBA data types we tested, the `chars` and `octets` performed the best. This is not surprising, and results from the fact that presentation layer conversions and data copying can be minimized for this "untyped" data (though both ORBs do perform multiple data copies for `chars` and `octets`). Throughput initially increases as the sender buffer size increases due to a decrease in the number of `writes` and `reads`. However, the increased throughput begins to level off once network-layer fragmentation occurs. For our ATM network, the maximum transfer unit (MTU) was 9,180 bytes. Thus, fragmentation occurs once the write buffers become greater than the MTU size. Although the MTU is 9,180 bytes, we do not observe the highest throughput for sender buffer sizes of 8K. For these buffer sizes, the number of `writes` is large and the presentation layer overhead plays a major role in decreasing the throughput.

Table 3 depicts the time spent by the sender and receiver of the ORBeline version of CORBA-TTCP using DSI when transferring 64 Mbytes of sequences using 128 K sender and receiver buffers and 64 K socket queues. An analysis for `octets` (best throughput) and `structs` (worst throughput) is presented. These figures precisely pinpoint the time spent in marshalling, data copying, and reading and writing of buffers. The time taken for `writes` suggests the use of non-optimal buffer management and flow control methods used by the ORBs.

[9] Oneway transfer results have not been presented since the ORBeline 2.0 frequently drops `oneway` operations at the server. Although this behavior is allowed by the "best effort" semantics of CORBA `oneway` operations, it appears to be a bug in the ORBeline implementation.

The analysis of the performance of the CORBA implementations suggests that optimized read/writes, presentation layer conversions and data copying are the primary areas that must be optimized to achieve higher throughputs.

4 Related Work

Existing research in gigabit networking has focused extensively on enhancements to TCP/IP. Less attention has been paid to integrating the following topics related to communication middleware like CORBA:

• **Transport Protocol Performance over ATM networks:** [4, 7, 2] present results on performance of TCP/IP (and UDP/IP [2]) on ATM networks by varying a number of parameters (such as TCP window size, socket queue size, and user data size). This work indicates that in addition to the host architecture and host network interface, parameters configurable in software (like TCP window size, socket queue size and user data size) significantly affect TCP throughput. [2] also shows that UDP performs better than TCP over ATM networks, which is attributed to redundant TCP processing overhead on highly-reliable ATM links.

A comparison of our current results for typed data with other work using untyped data in a similar CORBA/ATM testbed [13] reveal that the performance of Orbix for sequences of scalar data types is almost the same as that reported for untyped data sequences using the statically generated stubs and the IIOP. However, the performance of transferring sequences of CORBA `structs` for 64 K and 8 K was much worse than those for the scalars. As discussed earlier, this overhead arises from the amount of time the CORBA implementations spend performing presentation layer conversions and data copying.

• **Presentation layer and data copying:** The presentation layer is a major bottleneck in high-performance communication subsystems [1]. This layer transforms typed data objects from higher-level representations to lower-level representations (marshalling) and vice versa (demarshalling). In both RPC toolkits and CORBA, this transformation process is performed by client-side stubs and server-side skeletons that are generated by interface definition language (IDL) compilers. IDL compilers translate interfaces written in an IDL (such as XDR [15], NDR [3], or CDR [9]) to other forms such as a network wire format. A significant amount of research has been devoted to developing efficient stub generators. We cite a few of these and classify them as below.

- *Annotating high level programming languages* – The Universal Stub Compiler (USC) [10] annotates the C programming language with layouts of various data types. The USC stub compiler supports the automatic generation of device and protocol header marshalling code. The USC tool generates optimized C code that automatically aligns data structures and performs network/host byte order conversions.
- *Generating code based on Control Flow Analysis of interface specification* – [6] describes a technique of exploiting application-specific knowledge contained in the type specifications of an application to generate optimized marshalling code. This work tries to achieve an optimal tradeoff between interpreted code (which is slow but compact in size) and compiled code (which is fast but larger in size). A frequency-based ranking of application data types is used to decide between interpreted and compiled code for each data type.

None of the systems described above are targeted for communication middleware (*e.g.,* CORBA) requirements and constraints such as heterogeneity, high system reliability, efficient marshalling/demarshalling of parameters, flexible and efficient object location and selection, and higher level mechanisms for collaboration among services [8].

5 Concluding Remarks

An important and growing class of applications (such as interface browsers, network management applications, and distributed visualization/debugging tools) require flexible, high-performance communication middleware linked by high-speed networks. Contemporary CORBA implementations do not yet meet these requirements due to overhead stemming from (1) excessive presentation layer conversions and data copying, (2) inefficient server demultiplexing techniques, (4) long chains of intra-ORB function calls, and (4) non-optimized buffering algorithms used for network reads and writes. On low-speed networks this overhead is often masked. On high-speed networks, this overhead becomes a significant factor limiting communication performance and ultimately adoption by developers.

As users and organizations migrate to networks with gigabit data rates, the inefficiencies of current communication middleware (like CORBA) will force developers to choose lower-level mechanisms (like sockets) to achieve the necessary transfer rates thereby increasing development effort and reducing system reliability, flexibility, and reuse. This is a serious problem for mission/life-critical applications (such as satellite surveillance and medical imaging [4]).

This paper illustrates how existing CORBA implementations incur considerable overhead when application use the dynamic invocation interface (DII) and the dynamic skeleton interface (DSI) over high-speed ATM networks. Our DII and DSI results indicate that both Orbix 2.0 and ORBeline 2.0 incur different levels of presentation conversion overhead for different CORBA data types. Moreover, non-optimal internal buffer management strategies lead to large amounts of time spent in network writes and reads.

Finally, the experiments indicate that for DII, it is desirable to permit reuse of requests if the operations are `oneway` and the parameter values do not change. The current CORBA 2.0 specification does not define whether this is legal or not, so different implementations interpret the specification differently. Not only does this impede portability across ORB implementations, but it also permits non-optimal performance if requests are not reused.

Acknowledgements

We would like to thank IONA and Visigenic for their help in supplying the CORBA implementations used for these tests and for their help in fix bugs with the DII and DSI. Both companies are currently working to eliminate the performance overhead described in this paper and we expect their future releases to perform much better over high-speed networks.

Appendix

The following CORBA IDL interface was used for the Orbix and ORBeline CORBA implementations for both the DII and DSI tests:

```
struct PerfStruct{ short s; char c; long  l;
                   octet o; double d; };

// Richly typed data
interface ttcp_sequence
{
  typedef sequence<short> ShortSeq;
  // The rest of the sequence typedefs
  // are defined in a similar way.

  // Routines to send sequences of various data types.
  oneway void sendShortSeq (in ShortSeq ttcp_seq);

  // The rest of the methods are defined in a similar way.

  // To measure time taken for receipt of data.
  oneway void start_timer ();
  oneway void stop_timer ();
};
```

For the twoway data transfer case, the keyword `oneway` from the above interface was removed.

References

[1] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 200–208, Philadelphia, PA, September 1990. ACM.

[2] Sudheer Dharnikota, Kurt Maly, and C. M. Overstreet. Performance Evaluation of TCP(UDP)/IP over ATM networks. Department of Computer Science, Technical Report CSTR_94_23, Old Dominion University, September 1994.

[3] John Dilley. OODCE: A C++ Framework for the OSF Distributed Computing Environment. In *Proceedings of the Winter Usenix Conference*. USENIX Association, January 1995.

[4] Minh DoVan, Louis Humphrey, Geri Cox, and Carl Ravin. Initial Experience with Asynchronous Transfer Mode for Use in a Medical Imaging Network. *Journal of Digital Imaging*, 8(1):43–48, February 1995.

[5] Aniruddha Gokhale and Douglas C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. In *Proceedings of SIGCOMM '96*, Stanford, CA, August 1996. ACM.

[6] Phillip Hoschka and Christian Huitema. Automatic Generation of Optimized Code for Marshalling Routines. In *IFIP Conference of Upper Layer Protocols, Architectures and Applications ULPAA'94*, Barcelona, Spain, 1994. IFIP.

[7] K. Modeklev, E. Klovning, and O. Kure. TCP/IP Behavior in a High-Speed Local ATM Network Environment. In *Proceedings of the* 19^{th} *Conference on Local Computer Networks*, pages 176–185, Minneapolis, MN, October 1994. IEEE.

[8] Object Management Group. *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 edition, March 1995.

[9] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, July 1995.

[10] Sean W. O'Malley, Todd A. Proebsting, and Allen B. Montz. USC: A Universal Stub Compiler. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, London, UK, August 1994.

[11] Guru Parulkar, Douglas C. Schmidt, and Jonathan S. Turner. a^It^Pm: a Strategy for Integrating IP with ATM. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*. ACM, September 1995.

[12] Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt. Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging. In Douglas C. Schmidt, editor, *USENIX Computing Systems*. MIT Press, November/December 1996.

[13] Douglas C. Schmidt, Timothy H. Harrison, and Ehab Al-Shaer. Object-Oriented Components for High-speed Network Programming. In *Proceedings of the* 1^{st} *Conference on Object-Oriented Technologies and Systems*, Monterey, CA, June 1995. USENIX.

[14] Pure Software. *Quantify User's Guide*, 1995.

[15] Sun Microsystems. XDR: External Data Representation Standard. *Network Information Center RFC 1014*, June 1987.