Measuring the Performance of Communication Middleware on High-Speed Networks

Aniruddha Gokhale and Douglas C. Schmidt

gokhale@cs.wustl.edu and schmidt@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA

This paper will appear in the SIGCOMM Conference, 1996, Stanford University, August, 1996.

Abstract

Conventional implementations of communication middleware (such as CORBA and traditional RPC toolkits) incur considerable overhead when used for performance-sensitive applications over high-speed networks. As gigabit networks become pervasive, inefficient middleware will force programmers to use lower-level mechanisms to achieve the necessary transfer rates. This is a serious problem for mission/life-critical applications (such as satellite surveillance and medical imaging).

This paper compares the performance of several widely used communication middleware mechanisms on a high-speed ATM network. The middleware ranged from lower-level mechanisms (such as socket-based C interfaces and C++ wrappers for sockets) to higher-level mechanisms (such as RPC, hand-optimized RPC and two implementations of CORBA – Orbix and ORBeline). These measurements reveal that the lower-level C and C++ implementations outperform the CORBA implementations significantly (the best CORBA throughput for remote transfer was roughly 75 to 80 percent of the best C/C++ throughput for sending scalar data types and only around 33 percent for sending structs containing binary fields), and the hand-optimized RPC code performs slightly better than the CORBA implementations. Our goal in precisely pinpointing the sources of overhead for communication middleware is to develop scalable and flexible CORBA implementations that can deliver gigabit data rates to applications.

Keywords: Communication middleware, distributed object computing, CORBA, high-speed networks.

1 Introduction and Motivation

Despite dramatic increases in the performance of networks and computers, designing and implementing flexible and efficient communication software remains hard. Substantial time and effort has traditionally been required to develop this type of software; yet all too frequently communication software fails to achieve its performance and functionality requirements. Communication middleware based on the Common Object Request Broker Architecture (CORBA) [13] is a promising approach for improving the flexibility, reliability, and portability of communication software. CORBA is designed to enhance distributed applications by automating common networking tasks such as parameter marshalling, object location and object activation.

Experience over the past several years [19] indicates CORBA is well-suited for request/response applications over lower-speed networks (such as Ethernet and Token Ring). However, earlier studies [18, 23], and our results shown in Section 3, demonstrate that conventional implementations of CORBA incur considerable overhead when used for performance-sensitive applications over high-speed networks. 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. The use of low-level mechanisms increases development effort and reduces system reliability, flexibility, and reuse. This is a serious problem for mission/life-critical applications (such as satellite surveillance and medical imaging [3, 7]). Therefore, it is imperative that performance of high-level, but inefficient, communication middleware be improved to match that of low-level, but efficient, tools.

The primary contribution of this paper is to pinpoint precisely where the key sources of overhead exist in higher-level communication middleware such as CORBA and RPC toolkits. Our findings indicate that this overhead stems from a variety of sources including non-optimized presentation layer conversions, data copying, and memory management, generation of non-word boundary aligned data structures by the RPC and CORBA stub compilers, excessive control information carried in request messages, as well as inefficient and inflexible receiver-side demultiplexing and dispatching operations. Our goal in precisely pinpointing the sources of overhead for communication middleware is to develop scalable and flexible CORBA implementations that can deliver gigabit data rates to applications.

The paper is organized as follows: Section 2 outlines the CORBA communication middleware 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.

2 Overview of CORBA

CORBA is an open standard for distributed object computing [13]. The CORBA standard defines a set of components that allow client applications to invoke operations (`op`) with arguments (`args`) on object implementations. Flexibility is enhanced by using CORBA since the object implementations can be configured to

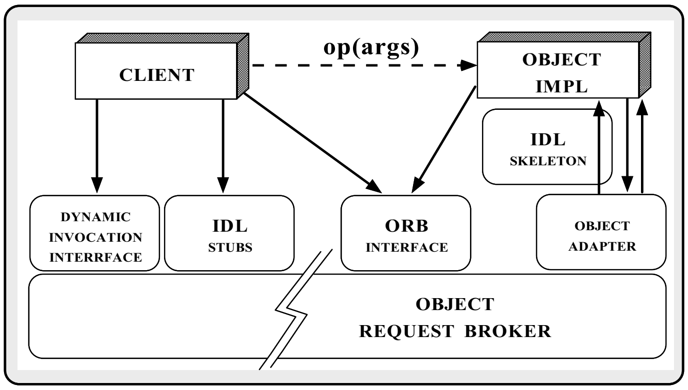

Figure 1: Components in the CORBA Distributed Object Computing Model

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. This interface 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:** CORBA IDL stubs and skeletons 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 CORBA IDL compiler. The use of a compiler greatly reduces the potential for inconsistencies between client stubs and server skeletons.

- **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 IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking *deferred synchronous* (separate send and receive operations) and *oneway* (send-only) calls.

- **Object Adapter:** assists the ORB with delivering requests to the object and with activating the object. More importantly, an object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects).

The use of CORBA as communication middleware enhances application flexibility and portability by automating many common development tasks such as object location, parameter marshalling, and object activation. CORBA is an improvement over conventional procedural RPC middleware (such as OSF DCE and ONC RPC) since it supports object-oriented language features (such as encapsulation, interface inheritance, parameterized types, and exception handling) and advanced design patterns [8] for distributed communication. These features and patterns enable complex distributed and concurrent applications to be developed more rapidly and correctly.

As shown below, the primary drawback to using higher-level middleware like CORBA is its poor performance over high-speed networks. In general, existing implementations of CORBA have not been optimized since performance has not been a problem on low-speed networks. It is beyond the scope of this paper to discuss limitations with CORBA features (see [2] for a synopsis).

3 Experimental Results of CORBA over ATM

This section describes our CORBA/ATM testbed and presents the results of our performance experiments.

3.1 CORBA/ATM Testbed Environment

3.1.1 Hardware and Software Platforms

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 5.4. The LattisCell 10114 is a 16 Port, OC3 155 Mbs/port switch. Each SPARCstation 20 contains two 70 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.4 TCP/IP protocol stack is implemented using the `STREAMS` communication framework [21]. 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 was measured at 1.4 Gbps, which is roughly comparable to an OC24 gigabit ATM network [16].

3.1.2 Traffic Generators

Earlier studies [23, 18] tested the performance of "flooding models" that transferred untyped bytestream data between hosts using several implementations of CORBA and other lower-level mechanisms like sockets. Untyped bytestream traffic is representative of applications like bulk file transfer and videoconferencing. Note, however, that bytestream traffic does not adequately test the overhead of presentation layer conversions since untyped data need not be marshalled or demarshalled. Ironically the implementations of CORBA used in our tests perform marshalling and demarshalling even for untyped `octet` data [23], which is further evidence that they have not been optimized for high-speed networks.

The experiments conducted for this paper extend our earlier studies [23] by measuring the performance of sockets, ACE C++ wrappers for sockets [22], an unoptimized and a hand optimized version of Sun's Transport Independent RPC (TI-RPC) [25], and two widely used implementations of CORBA (Orbix 2.0 and ORBeline 2.0) to transfer both bytestream and typed data between remote hosts over a high-speed ATM network. The use of typed data is representative of applications like electronic medical imaging [7, 3] and high-speed distributed databases (such as global change repositories [17]). In addition, measuring typed data transfer reveals the overhead of presentation layer conversions and data copying for the various communication middleware mechanisms we measured.

Traffic for the experiments was generated and consumed by an extended version of the widely available TTCP protocol benchmarking tool. We extended TTCP for use with C sockets, C++ socket wrappers, TI-RPC, Orbix, and ORBeline. Our TTCP tool measures end-to-end data transfer throughput in Mbps from a transmitter process to a remote receiver process across an ATM network or host loopback. The flow of user data for each version of TTCP is uni-directional, with the transmitter flooding the receiver with a user-specified number of data buffers. Various sender and receiver parameters may be selected at run-time. These parameters 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 C++ `struct` composed of all the scalars (`BinStruct`). The CORBA implementation transferred the data types using IDL `sequences`, which are dynamically-sized arrays. To compare CORBA with C and C++, we defined `structs` in the same manner that the CORBA IDL compiler generated `sequences`. Likewise, to compare CORBA with TI-RPC, we generated `structs` using unbounded arrays defined in the RPC language (RPCL). These definitions are shown in the Appendix.

The C and C++ versions of TTCP were written using the standard Internet family of macros that convert values between host and network byte order. These macros are implemented as "no-ops" because the sender and receiver processes both ran on SPARCs, which use big-endian network byte order. Therefore, the C/C++ versions do not actually perform any presentation layer conversions on the data. The CORBA and the RPC versions of TTCP also omit these conversions since they use the byte order macros, as well. However, the CORBA and RPC implementations do *not* omit the overhead of the function calls, which has a non-trivial overhead (shown in Section 3.2.2).

3.1.3 TTCP Parameter Settings

Existing studies [7, 11, 6, 23, 18] of transport protocol performance over ATM demonstrate the impact of parameters such as socket queue sizes and data buffer on performance. Therefore, our TTCP benchmarks varied these two parameters for each type of data as follows:

- **Socket queue size:** the sender and receiver socket queue sizes used were 8 K and 64 K bytes (on SunOS 5.4 these are the default and maximum, respectively). These parameters influence the size of the TCP segment window, which has been shown to significantly impact CORBA-level and TCP-level performance on high-speed networks [11, 23]. Since the performance of the 8 K socket queues was consistently one-half to two-thirds slower than using the 64 K queues, we omitted the 8 K results from the figures below.

- **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.

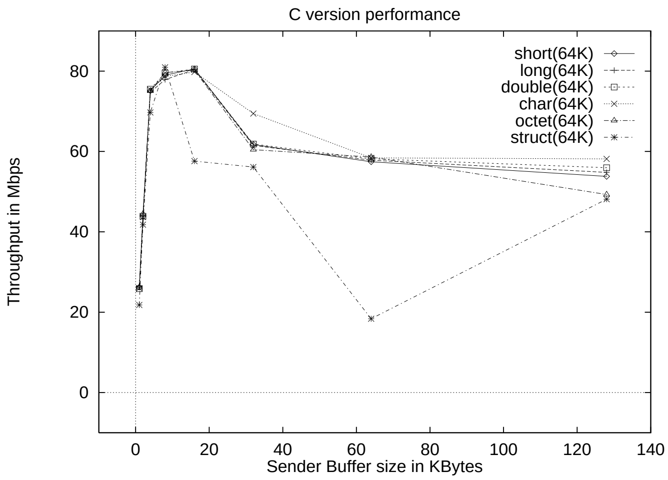

Figure 2: Performance of the C Version of TTCP

3.2 Performance Results

The performance results from our experiments are reported below. The throughput measurements for each of the six versions of TTCP are presented first. Detailed profiling measurements of presentation layer, data copying, demultiplexing, and memory management overhead are presented in Sections 3.2.2 and 3.2.3. The profile information was obtained using the `Quantify` performance measurement tool. `Quantify` analyzes performance bottlenecks and identifies sections of code that dominate execution time. An important feature of `Quantify` is its ability to report results without including its own overhead, unlike traditional sampling-based profilers like the UNIX `gprof` tool.

3.2.1 Throughput Measurements

Remote Transfer Results: Figures 2, 3, 6, 7, 8 and 9 depict the throughput obtained for sending 64 MB data of various data types for each TTCP implementation over ATM. These figures present 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 5.4).[1] This section analyzes the overall trends of the throughput for each communication middleware mechanism. Sections 3.2.2 and 3.2.3 use profiling output from `Quantify` to explain why performance differences occur.

- **C and C++ versions of TTCP:** Figures 2 and 3 indicate that the C and C++ versions both achieved a maximum of 80 Mbps throughput for sender buffer sizes of 8 K and 16 K bytes. The similarity between the results indicates that the performance penalty for using the higher-level C++ wrappers is insignificant, compared with using C socket library function calls directly.

As shown in the figures, the throughput increases steadily from 1 K to 8 K buffer sizes. The reason for this is that as the sender buffer size increases, the sender requires fewer `writes` to transmit 64 MB of data. The throughput peaks between the 8 K and 16 K buffer sizes and then gradually decreases – leveling off at around

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

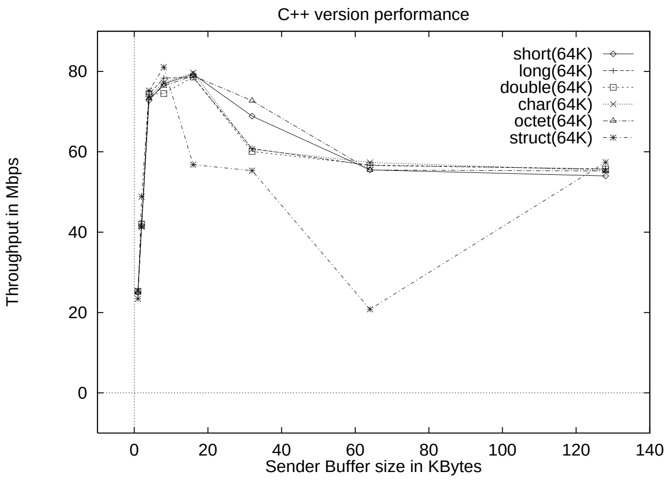

Figure 3: Performance of the C++ Wrappers Version of TTCP

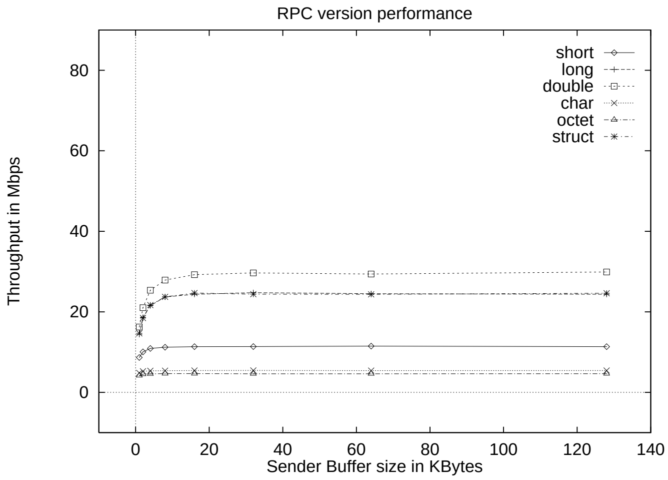

Figure 6: Performance of the RPC Version of TTCP

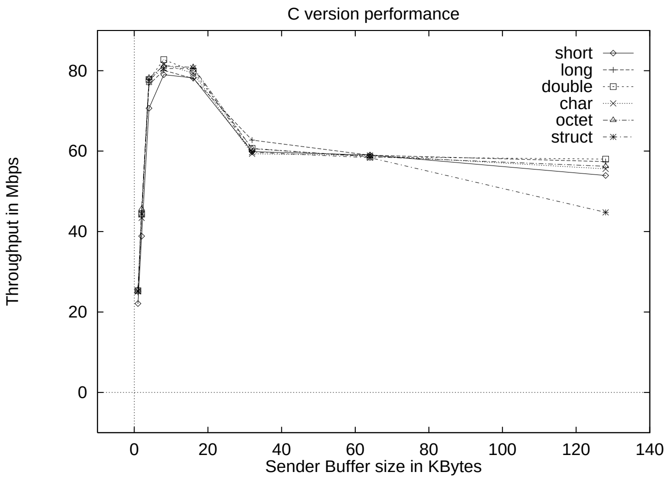

Figure 4: Performance of the Modified C Version of TTCP

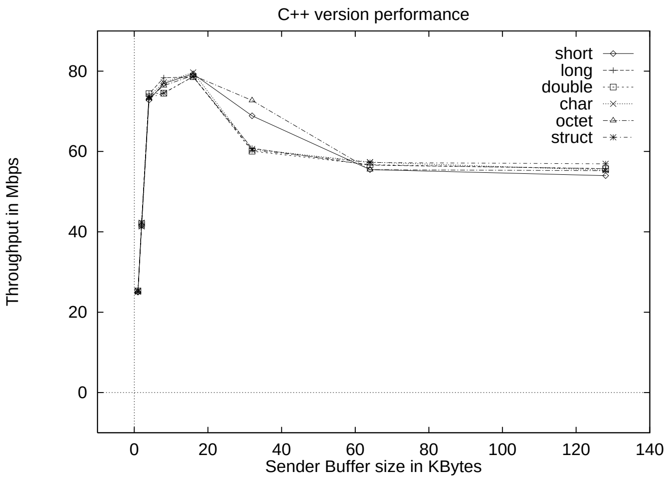

Figure 5: Performance of the Modified C++ Version of TTCP

60 Mbps. This drop off between 8 K and 16 K arises from the 9,180 MTU of the ATM network. When sender buffer sizes exceed this amount, fragmentation at the IP and ATM driver layers degrades performance. As sender buffer sizes increase, fragmentation becomes a dominant factor, yielding the performance curves shown in Figures 2 and 3.

Close scrutiny of Figures 2 and 3 illustrate unusual behavior for `BinStructs` when the sender buffers are 16 K and 64 K. In these cases throughput drops sharply. Analysis of `Quantify's` profile information for 64 K sender buffers revealed that the `writev` system call is called 1,025 times, accounting for 28,031 msec of the total execution time. In contrast, in the best case (sending `longs`) the 1,025 calls to `writev` accounted for only 9,087 msec of the total execution time.

This aberrant behavior occurs since 64 K is not an integral multiple of the size of the C and C++ `BinStruct` data type (which is 24 bytes). Therefore, the sender buffers were slightly less than 64 K when written with the `writev` function. This minor difference apparently triggered interactions between the SunOS 5.4 internal STREAMS buffering strategy and the TCP sliding window protocol, which yielded extremely low throughput. To work around this problem, we defined a C/C++ `union` that ensures the size of the transmitted data is rounded up to the next power of 2 (in this case 32 bytes). This enabled TTCP to send 64 K bytes in a single `writev` call and obtain throughput comparable to the other data types. These new results are shown in figure 4 and 5.

- **RPC version of TTCP:** Figures 6 and 7 show the performance of the original and hand-optimized RPC versions of TTCP. The original stubs generated automatically by RPCGEN attained extremely low throughout (peaking at 29 Mbps for `doubles`, which is only 35% of the throughput attained by the C and C++ versions). `Quantify` analysis reveals that this poor performance was due to excessive data copying and presentation layer conversions performed by XDR (explained in Section 3.2.2).

To make the implementation comparable to the C/C++ TTCP implementations, we hand-optimized the RPC generated code for TTCP. For all the data types, the `xdr_bytes` function generated by RPCGEN was used to send/receive data. This avoided the overhead of converting between the native and XDR formats. This optimization was valid because the data was transferred between big-endian SPARCstations with the same alignment and word length.

The hand-optimized code improved the performance significantly. Figure 7 illustrates the results are 79% of the C and C++

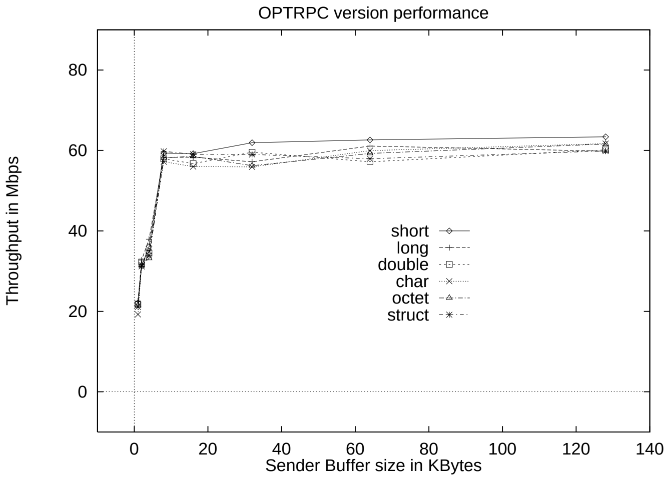

Figure 7: Performance of the Optimized RPC Version of TTCP

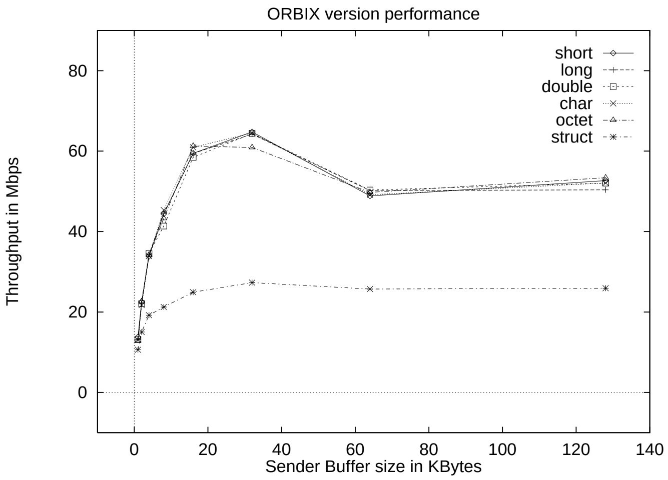

Figure 8: Performance of the Orbix Version of TTCP

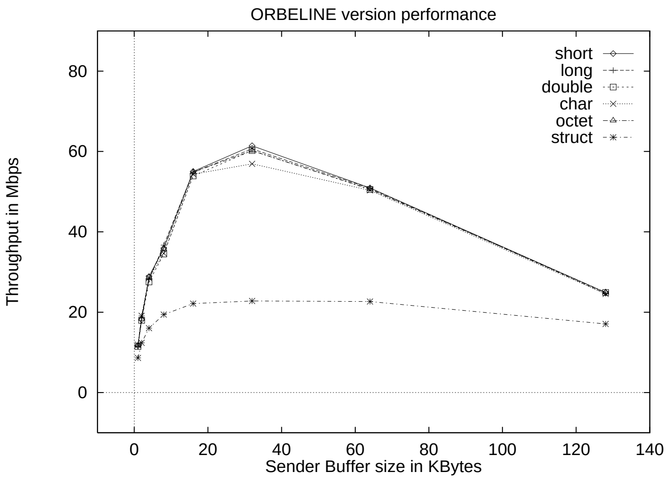

Figure 9: Performance of the ORBeline Version of TTCP

performance. These results indicate that for sender buffer sizes from 8 K to 128 K, the measured throughput was roughly 59-63 Mbps. The throughput steadily increases until the sender buffer size reaches 8 K. Beyond this point there was only a marginal improvement in the throughput.

The shape of the optimized RPC performance curves result from the 9,000 bytes data buffer sizes sent by the generated RPC stubs. Analysis from `Quantify` and the SunOS5.4 system-call tracing command (`truss`) reveals that the RPC sender-side stubs use 9,000 byte internal buffers to make the `writes`. As a result, the performance attained for sender buffer sizes from 8 K to 128 K show only a marginal improvement, which is attributed to the use of 64 K socket queue sizes maintained by the sender and receiver.

• **CORBA versions of TTCP:** Figures 8 and 9 illustrate the throughput obtained for both CORBA implementations. These figures indicate that the throughput steadily increases until the sender buffers reach 32 K, at which point it peaks at 65 Mbps for Orbix and 60 Mbps for ORBeline for sending scalars. Beyond this point, performance gradually decreases. This behavior differs from the C and C++ versions, which peak at 8 K and 16 K. ORBeline performance falls off much more quickly than Orbix performance. This effect is noticeable for sender buffer size of 128 K in Figure 9.

Analysis using `truss` for 128 K sender buffer size revealed that both the Orbix and the ORBeline versions try to write the entire 128 K bytes plus some control information (56 bytes for Orbix and 64 bytes for ORBeline). The Orbix version uses the `write` system call, whereas the ORBeline version uses the `writev` system call.

Analysis using `Quantify` indicated that to send 64 MB user data using 128 K user buffer size, the Orbix version attempted a total of 538 `writes`, which required 9,638 msec. In contrast, the ORBeline version made a total of 512 `writevs`, which required 20,319 msec to complete. This explains the lower throughput for the ORBeline client. The receiver performance in both cases is comparable. The `truss` output for the Orbix and the ORBeline receiver shows that the time spent by ORBeline in `reads` is marginally smaller than that of the Orbix version, but this is offset by the 4,252 `poll` system calls made by ORBeline compared to only 539 made by the Orbix receiver.

For the 32 K data buffers, the performance of Orbix and ORBeline is comparable with the 65-70 Mbps attained by the C/C++ versions. Likewise, the optimized RPC version achieved roughly the same throughput as the CORBA implementations. However, both CORBA implementations achieved approximately half the throughput for `structs`. As shown in Section 3.2.2, this performance reduction occurs from the high amount of presentation layer conversions and data copying in Orbix and ORBeline. In addition, `truss` revealed that both the CORBA implementations write buffers containing only 8 K when sending `structs`. In contrast, for 32 K data buffers, they sent scalars in buffers containing all 32 K data plus additional control information, as described above. This behavior adds to the overhead imposed by data copying and presentation layer conversions and greatly reduces the performance.

Loopback Results: Figures 10, 11, 12, 13, 14 and 15 depict the throughput obtained by replicating the TTCP tests described above over the SPARCstation loopback device. Measuring loopback behavior approximates the performance of communication middleware for channel speeds greater than our 155 Mbps ATM network.

• **C/C++ Results:** The results indicate that for C/C++ versions of TTCP, the throughput starts leveling off around 8 K sender buffer size at roughly 190-197 Mbps. Due to the implementation of the loopback device in SunOS 5.4, throughput was not affected as significantly by fragmentation overhead compared with the ATM results shown in Figures 2 and 3.

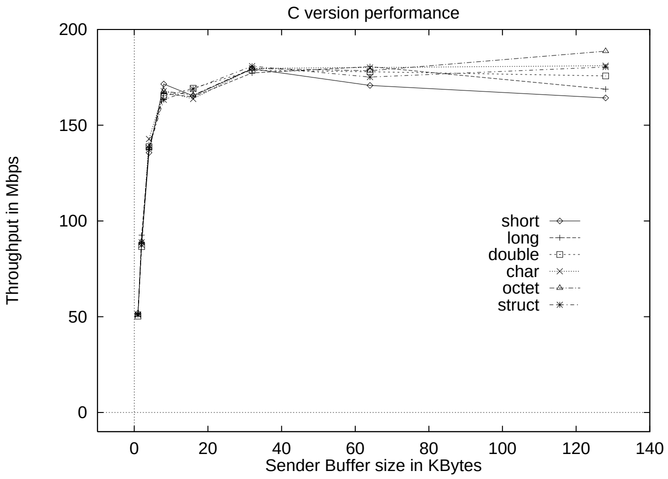

Figure 10: Performance of the C Loopback Version of TTCP

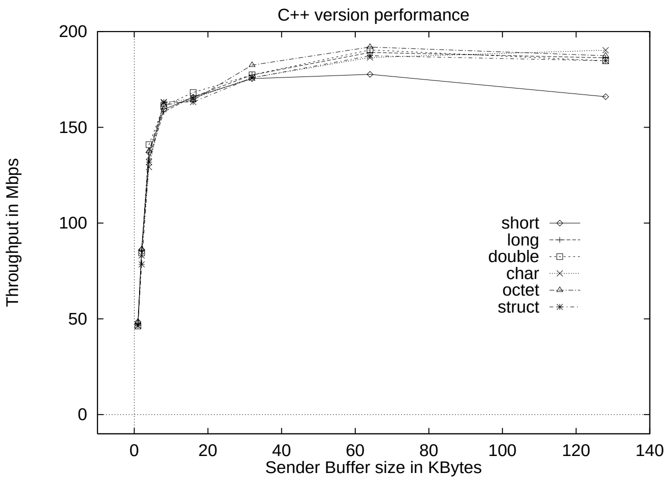

Figure 11: Performance of the C++ Wrappers Loopback Version of TTCP

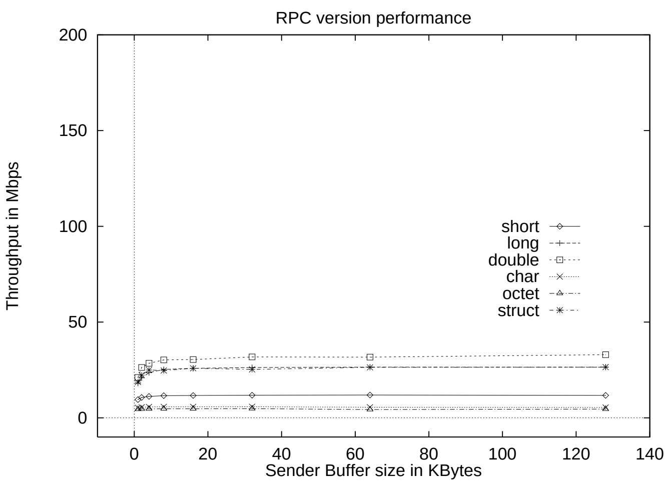

Figure 12: Performance of the RPC Loopback Version of TTCP

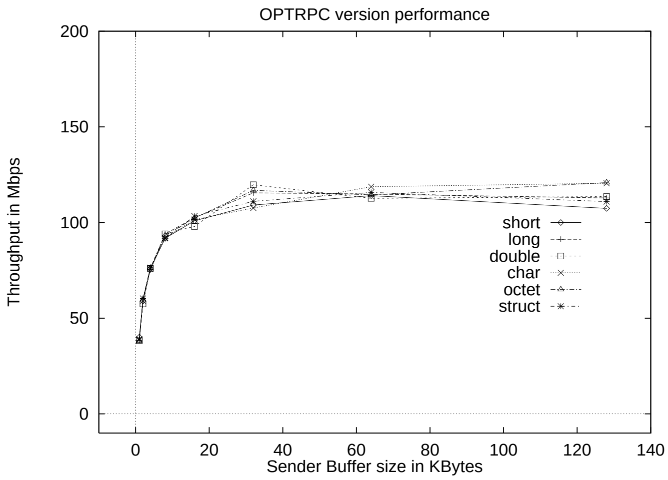

Figure 13: Performance of the Optimized RPC Loopback Version of TTCP

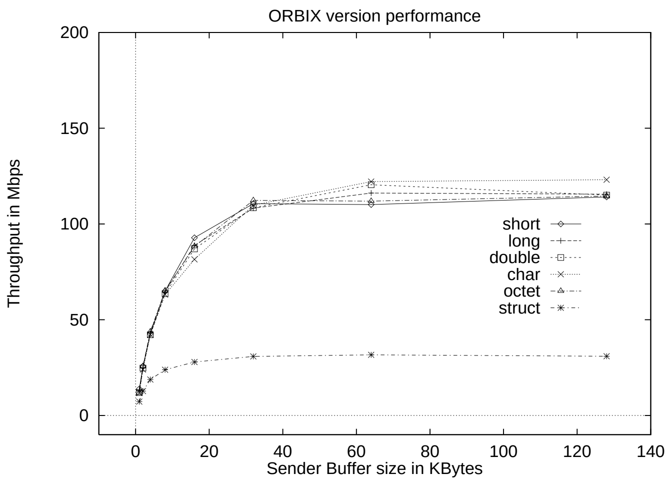

Figure 14: Performance of the Orbix Loopback Version of TTCP

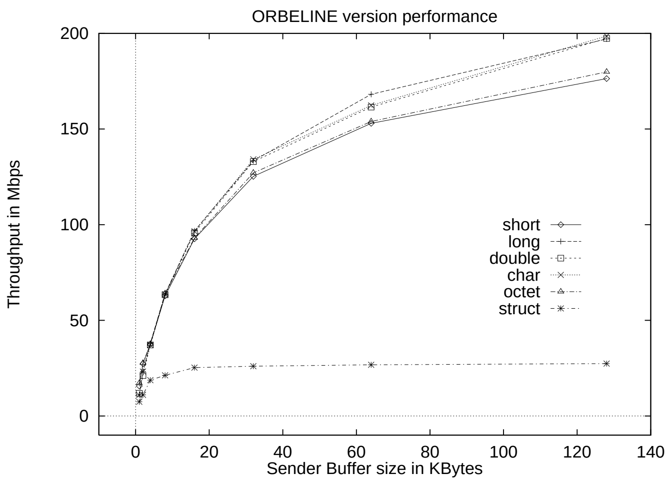

Figure 15: Performance of the ORBeline Loopback Version of TTCP

TTCP version	Remote Transfer				Loopback			
	Scalars		Struct		Scalars		Struct	
	Hi	Lo	Hi	Lo	Hi	Lo	Hi	Lo
C/C++	80	25	80	25	197	47	190	47
Orbix	65	15	27	11	123	14	32	10
Orbeline	61	12	23	9	197	11	27	7
RPC	30	4	25	14	33	5	27	18
optRPC	63	20	63	20	121	38	116	38

Table 1: Summary of Observed Throughput for Remote and Loopback Tests in Mbps

• **RPC Results:** The original RPC version did not show any significant change over the remote transfer results. The optimized RPC version of TTCP exhibited behavior similar to C/C++ over the loopback, leveling off at around 110-115 Mbps. This behavior is attributed to the smaller internal buffer size [2] RPC uses to `write` data on the sender-side and `read` on the receiver-side. This smaller size increases the number of times these functions are invoked.

• **CORBA Results:** The Orbix version of TTCP behaves like the optimized RPC for all scalar data types. The ORBeline version shows a gradual increase in throughput, which peaks at around 197 Mbps for a sender buffer size of 128 K which is close to the C/C++ version performance for the loopback case. Analysis using `Quantify` for 128 K sender buffers reveals that both Orbix and ORBeline senders and receivers spend approximately equal amount of time in `writes` and `reads`. However, the Orbix version spends around 896 msec in `memcpy` on both the sender and receiver side compared to only 1.51 msec for ORBeline sender and 15.51 msec for ORBeline receiver.

This explains why the ORBeline throughput for loopback is higher than the Orbix throughput with increasing buffer size. However, both the Orbix and ORBeline still perform poorly for `structs` because they spend a significant amount of time performing presentation layer conversions and data copying. Although ORBeline provides an option of using shared memory, we did not use this option since our goal is to measure throughput over the network and not the memory speed.

In general, the highest throughput for Orbix is approximately 75-80% that of the C/C++ versions for remote transfers and around 68% for the loopback test. The difference in the throughputs is most apparent for `structs`. In this case, the throughput for both the Orbix and ORBeline versions is roughly 33% of the C/C++ version for remote transfers and 16% for the loopback tests.

These findings illustrate that as channel speeds increase, the performance of the CORBA implementations become worse relative to that achieved by low-level communication middleware since the overhead of presentation layer conversions and data copying become increasingly dominant. Thus, it is imperative that such overhead be eliminated so that CORBA can be used effectively to build flexible and reliable middleware capable of delivering very high data rates to applications.

The results for the remote and loopback tests for all versions of TTCP are summarized in Table1. This table depicts the highest and the lowest throughput attained by each version of TTCP for all the scalars and `structs`. In addition, we combine results for the C and C++ versions of TTCP since their performance is similar. All entries are in Mbps rounded up to the nearest integer.

3.2.2 Presentation Layer and Data Copying Overhead

[2] As explained earlier, the RPC version used an internal buffer of roughly 9,000 bytes for writing and reading. In contrast, the C/C++ versions used a 64 KB read buffer and sends are done according to the size of buffers passed by the client.

TTCP Version	Data Type	Analysis		
		Method Name	msec	%
C/C++	struct	writev	9,415	98
RPC	char	write	283,350	89
		xdr_char	17,000	5
	short	write	134,855	90
	long	write	71,600	92
	double	write	37,877	87
		xdr_double	2,348	5
	struct	write	80,517	92
optRPC	struct	write	4,262	80
		memcpy	896	17
Orbix	char	write	9,638	89
		memcpy	895	8
	struct	write	26,366	68
		NullCoder::codeLongArray	1,162	3
		BinStruct::encodeOp	952	2
		CHECK	932	2
		Request::encodeLongArray	812	2
		Request::insertOctet	782	2
		Request::op<<(double&)	838	2
		Request::op<<(short&)	782	2
		Request::op<<(long&)	782	2
		Request::op<<(char&)	782	2
ORBeline	char	writev	20,319	99
	struct	writev	82,794	84
		op<<(NCostream&,BinStruct&)	3,831	4
		memcpy	3,594	4
		PMCIIOPStream::put	951	1
		PMCIIOPStream::op<<(double)	978	1
		PMCIIOPStream::op<<(long)	950	1

Table 2: Sender-side Overhead

Table 2 and 3 depict the time spent by the senders and receivers of various versions of TTCP when transferring 64 Mbytes of sequences using 128 K sender and receiver buffers and 64 K socket queues. For each version, an analysis for a specific data type is presented if it resulted in throughput that differs from that of the rest of the data types. Otherwise, an analysis for a representative data type is presented (*e.g.,* `BinStruct`).

For Orbix or ORBeline, `structs` resulted in throughput that differed significantly from the throughput for the rest of the data types. Therefore, we present analysis for `struct` and `char`, which are the representative data types. In the tables, the % column shows the percentage of the total execution time attributed to the corresponding function under the *Method Name* column. The time spent in milliseconds by this method is indicated in the *msec* column. This fine-grained profiling information reveals precisely why the C and C++ implementations outperform the RPC and CORBA implementations.

Sender-side Overhead: The overhead for the sender-side presentation layer and data copying is presented below for each version of the TTCP benchmarks.

• **C/C++ Overhead:** The C and C++ versions of TTCP spent over 98% of their run-time making `writev` system calls. In this case, there is no presentation layer conversion overhead. As explained earlier, the standard Internet family of macros that convert values between host and network byte order are implemented as "no-ops".

• **RPC Overhead:** The RPC version of TTCP spends different amounts of time writing various data types. For instance, to write chars, the RPC version takes 283,330 msec compared to 71,600 msec for writing `longs`. The reason for this behavior is due to the RPC XDR mapping, which converts a single byte `char` into a four byte data representation before it is sent over the network. The hand-optimized version of RPC considers all data types as `opaque`, which avoids the XDR mapping for each data type. The hand-optimized RPC version of TTCP is also largely `write` bound, though it spends about 17% of its time performing data copies with memcpy. The significant amount of memcpys is due to the XDR routine xdrrec_putbytes being called many times on the sender-side. The user buffer is copied into an internal buffer, which is then sent over the network.

• **CORBA Overhead:** The sender-side of the Orbix version of TTCP sending BinStructs spent a significant amount of time marshalling the fields that made up the BinStruct. For 64 MB of data and a sender buffer of 128 KB, the client invokes the sendBinStruct method 512 times. This method invokes the IDL compiler generated _IDL_SEQUENCE _BinStruct ::encodeOp method. Since a BinStruct is 32 bytes, each sender buffer of size 128 KB can accommodate 4,096 structs. For each struct, the Orbix version marshalled each field using CORBA::Request methods like ::encodeLongArray and ::operator<< (const long&). Each of these marshalling routines was invoked for the 4,096 structs in a single buffer for 512 iterations, yielding a total of 2,097,152 invocations!

Analysis using Quantify revealed that for BinStructs, the Orbix sender spent around 68% of its time (26,366 msec) in writes, 1.71% (671 msec) doing memcpys and over 18% time marshalling the structure. Likewise, the ORBeline sender spent around 84% of the time (82,724 msec) in writevs, with approximately 4% in memcpy and 10% marshalling the structure.

Receiver-side Overhead: Our benchmarks found the receiver-side tests performed similar to the sender-side tests.

• **C/C++ Overhead:** The C and C++ versions spend the bulk of their time in read and readv. The C and C++ versions on the receiver side used readv to read the length, type and buffer fields of the structures, thereby avoiding an intermediate copy. If the buffer is not completely received by readv, subsequent reads fill in the rest of the buffer.

• **RPC Overhead:** The receiver side analysis for the RPC version of TTCP shows that the RPC code spends a significant amount of time demarshalling various data types from the XDR network representation to the native host format. For instance, to demarshall the chars using xdr_char takes 30,422 msec. In contrast, to demarshall longs takes only 4,697 msec. As explained earlier, the hand-optimized RPC version eliminates this marshalling overhead by treating the data types as opaque.

The hand-optimized RPC version spent significant time doing getmsg, which stems from the use of the System V STREAMS in Sun's TI-RPC. Similar to the sender-side results in Table 2, the receiver-side RPC implementation spends about one-third of its time performing data copying. The significant amount of time spent in memcpy is due to a large number of calls to an internal function called get_input_bytes, which in turn is invoked by xdrrec_getbytes. The buffer received through calls to t_rcv is copied into another buffer, which is subsequently passed to the user application. The contribution of these functions to the total execution time is insignificant and hence their results are omitted from the table.

• **CORBA Overhead:** The results for Orbix indicate that a considerable amount of time was spent demarshalling each field of the structs that were received. This task was performed by a number of overloaded operator>> methods of the CORBA::Request class *e.g.,* CORBA::Request::operator>>(double &) to demarshall double types. The Quantify analysis of ORBeline's server-side to receive structs reveals that around 19% of the time was spent in memcpy, 18% in reads, and a large percent of its time in demarshalling the BinStructs.

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

TTCP Version	Data Type	Analysis: Method Name	msec	%
C/C++	struct	read	7,199	75
		readv	2,374	24
RPC	char	xdr_char	30,422	44
		xdrrec_getlong	16,998	24
		xdr_array	14,317	20
		getmsg	5,977	8
	short	xdr_short	11,184	36
		xdrrec_getlong	8,499	27
		xdr_array	7,158	23
		getmsg	2,969	9
	long	xdr_long	4,697	31
		xdrrec_getlong	4,250	28
		xdr_array	3,579	23
		getmsg	1,639	10
	double	xdr_double	3,467	29
		xdrrec_getlong	4,250	35
		xdr_array	1,790	15
		getmsg	1,562	13
	struct	xdrrec_getlong	4,250	26
		xdr_BinStruct	2,684	16
		getmsg	1,518	9
		xdr_char	1,267	7
		xdr_uchar	1,267	7
		xdr_double	1,155	7
optRPC	struct	getmsg	2,229	67
		memcpy	897	27
Orbix	char	read	7,915	85
		memcpy	896	9
	struct	read	4,280	26
		NullCoder::codeLongArray	1,314	8
		CHECK	923	5
		BinStruct::decodeOp	923	5
		Request::extractOctet	699	4
		Request::op>>(double&)	699	4
		Request::op>>(short&)	699	4
		Request::op>>(long&)	699	4
		Request::op>>(char&)	699	4
		memcpy	672	4
ORBeline	char	read	3,041	85
	struct	memcpy	3,581	19
		read	3,533	18
		op>>(NCistream&,BinStruct&)	3,495	18
		PMCIIOPStream::get	1,121	5
		PMCIIOPStream::op>>(double)	1,118	5
		PMCIIOPStream::op>>(long)	1,118	5

Table 3: Receiver-side Overhead

Function Name	Time in msec for Iterations			
	1	100	500	1,000
`strcmp`	3.89	376	1,882	3,764
`large_dispatch`	1.34	134	670	1,341
`ContextClassS::continueDispatch`	0.52	52	259	519
`ContextClassS::dispatch`	0.55	54	270	540
`FRRInterface::dispatch`	0.44	44	219	439
Total	6.74	660	3,300	6,603

Table 4: Server-side Demultiplexing Overhead in Orbix

3.2.3 Demultiplexing Overhead

A CORBA request message contains the identity of its remote object implementation and its intended remote operation. The remote object implementation is typically identified by an object reference and the operation is typically represented as a string or as a binary value. An ORB's Object Adapter is responsible for demultiplexing the request message to the appropriate method of the object implementation.

The type of demultiplexing scheme used by an ORB can impact performance significantly. Most ORBs (including ORBeline and Orbix) perform CORBA request demultiplexing in the following two steps:

1. *Object Adapter to IDL Skeleton* – the ORB uses the object reference in the request to locate the appropriate object implementation and associated IDL skeleton;
2. *IDL Skeleton to Implementation Method* – the IDL skeleton locates the appropriate method and performs an upcall, passing along the parameters in the request.

Performing two demultiplexing steps can be expensive, particularly when a large number of operations appear in an IDL interface.

To determine the cost of demultiplexing, we measured the server side request demultiplexing overhead for both the versions of CORBA. We defined an interface with a large number of methods (100 were used in this experiment). The method names were all unique. Four sets of results were obtained by running the client for 1, 100, 500 and 1,000 iterations. In each iteration, the client invoked the final method defined by the interface one hundred times, which evokes the worst-case behavior for Orbix because it uses linear search.

The results for Orbix are shown in Table 4.[3] In Orbix, the server executes an `impl_is_ready` function and waits for an event to occur. Whenever a request arrives, the server processes the event and invokes the `MsgDispatcher::dispatch` method. This method demultiplexes the request to the appropriate object implementation by calling the chain of dispatch methods shown in the Table.

The Orbix method `large_dispatch` inspects the operation name field of the arriving request. It makes a string comparison with each entry in the table of interfaces that the server maintains. The demultiplexing is based on the outcome of these string comparisons. Since the client test always sends the final method defined by the interface, the server side dispatching mechanism performs 100 string comparisons before it demultiplexes the incoming request. For a large interface, demultiplexing based on linear search is a significant bottleneck. In addition, passing operation names in every request adds to the overall control information which effectively reduces throughput as it occupies network bandwidth.

A better demultiplexing scheme would use hashing or direct indexing to reduce this bottleneck. This could reduce the control information overhead and also use efficient numeric comparisons,

[3]The function names appearing in the figure contribute to the incoming request demultiplexing and dispatching processing.

Function Name	Time in msec for Iterations			
	1	100	500	1,000
`atoi`	0.04	4	22	44
`large_dispatch`	0.52	52	260	520
`ContextClassS::continueDispatch`	0.52	52	259	519
`ContextClassS::dispatch`	0.55	54	270	540
`FRRInterface::dispatch`	0.44	44	219	439
Total	2.07	206	1,030	2,062

Table 5: Optimized Server-side demultiplexing in Orbix

Function Name	Time in msec for Iterations			
	1	100	500	1,000
`PMCSkelInfo::execute`	0.08	6	32	64
`PMCBOAClient::request`	0.51	51	253	507
`PMCBOAClient::processMessage`	0.48	47	235	471
`PMCBOAClient::inputReady`	0.43	42	209	417
`dpDispatcher::notify`	0.70	65	325	651
`dpDispatcher::dispatch`	0.43	40	201	401
Total	2.63	251	1,255	2,511

Table 6: Server-side Demultiplexing overhead in ORBeline

as opposed to string comparisons. To illustrate the benefits of these optimizations, we modified the CORBA stubs and skeletons by assigning unique values to each of the methods defined by our IDL test interface. In the request message, this unique number was passed as a string in place of the entire operation name. On the receiving side, the dispatcher performs an `atoi` to retrieve the number in numeric form. A direct indexing scheme based on a `switch` statement then performs a numeric comparison, which significantly improves demultiplexing performance by roughly 70%. The improvements obtained by this approach are shown in Table 5.

The ORBeline IDL compiler generated code uses inline hashing for server side demultiplexing of incoming requests. Table 6 shows the time spent by various functions contributing to the demultiplexing process. The server executes an `impl_is_ready` and waits for requests to arrive. As soon as a request arrives, this method invokes the `dispatch` method of the `dpDispatcher` class. This results in a chain of function calls shown in the table. Finally, the `PMCSkelInfo::execute` method invokes the appropriate method in the skeleton, which finally invokes the actual method in the object implementation.

The ORBeline code also passed method names in the requests along with other control information. To reduce the control information overhead, we used a similar optimization as in the case of Orbix, where all method names were assigned unique numeric values and these were passed as strings in the outgoing request message.[4]

Table 7 shows the time the client required for invoking the final method defined by the interface for the Orbix IDL compiler gener-

[4]The optimized version of ORBeline performed marginally better than the original and hence results for the demultiplexing overhead are omitted for the optimized case.

Version	Iterations			
	1	100	500	1,000
Original Orbix	0.274	25.99	130.57	263.70
Optimized Orbix	0.256	25.47	127.46	255.65
Original Orbeline	0.22	21.1	105.94	212.89
Optimized Orbeline	0.20	20.81	104.32	210.07

Table 7: Client-side Latency (in seconds) for sending 100 requests per iteration

Version	Iterations			
	1	100	500	1,000
Orbix	6.56	2.0	2.38	3.05
Orbeline	9.09	1.37	1.53	1.32

Table 8: Percentage improvement in client-side latency for sending 100 requests per iteration

Version	Iterations			
	1	100	500	1,000
Original Orbix	0.054	6.8	42.03	85.92
Optimized Orbix	0.049	4.86	36.94	76.94

Table 9: Client-side Latency (in seconds) for Sending 100 Requests per iteration using oneway methods

ated code, the optimized Orbix code, the ORBeline IDL generated code and the optimized ORBeline code. Table 8 shows the percentage improvement in latency due to optimizations for both the versions of CORBA. The table entries for the two versions of Orbix indicate that as the number of method invocations increases, the original Orbix code performs poorly as compared to the optimized code (*e.g.,* for 1000 iterations, the latency for the optimized Orbix code was 255.65 seconds compared to 263.7 seconds for the original Orbix code). Since ORBeline uses inline hashing for server side request demultiplexing, its performance is superior compared to that of Orbix by roughly 18-20%. The optimizations used with ORBeline reduced the amount of control information sent over the network, but it did not change the demultiplexing strategy used by the receiver. As a result, there was marginal improvement in the optimized ORBeline receiver's performance.

We performed the same experiment with Orbix using `oneway` methods in the interface definition. Table 9 shows the time the client required for invoking the final method defined by the interface for the Orbix IDL compiler generated code and the optimized Orbix code. Table 10 shows the percentage improvement in client-side latency for the `oneway` case. These tables indicate that improvement in latency for the `oneway` case due to optimizations was roughly 10% compared to only 3% for the `two-way` case. Since optimizations in ORBeline showed only marginal improvements in latency in the `two-way` case, we did not perform the `oneway` experiment for ORBeline.

4 Related Work

Existing research in gigabit networking has focused extensively on enhancements to TCP/IP. None of the systems described below are explicitly targeted for the requirements and constraints of communication middleware like CORBA. In particular, less attention has been paid to integrating the following topics related to communication middleware:

4.1 Transport Protocol Performance over ATM Networks

[7, 11, 6] present results on performance of TCP/IP (and UDP/IP [6]) 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. [6] also shows that UDP performs better than TCP over ATM networks, which is attributed to redundant TCP processing overhead on highly-reliable ATM links.

Version	Iterations			
	1	100	500	1,000
Orbix	9.25	28.52	12.11	10.45

Table 10: Percentage improvement in client-side latency for sending 100 requests per iteration using oneway methods

A comparison of our current results for typed data with other work using untyped data in a similar CORBA/ATM testbed [23] reveal that the low-level C socket version and the C++ socket wrapper versions of TTCP are roughly equivalent for a given socket queue size. Likewise, the performance of Orbix for sequences of scalar data types is almost the same as that reported for untyped data sequences. 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 in Section 3.2.2, this overhead arises from the amount of time the CORBA implementations spend performing presentation layer conversions and data copying.

4.2 Presentation Layer and Data Copying

The presentation layer is a major bottleneck in high-performance communication subsystems [5]. 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 Sun RPC XDR [24], DCE NDR, or CORBA CDR [13]) 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) [14] 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:** [10] 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. Our implementations of the stub compiler will be designed to adapt according to the runtime access characteristics of various data types and methods. The runtime usage of a given data type or method can be used to dynamically link in either the compiled or the interpreted version. Dynamic linking has been shown to be useful for mid-stream adaptation of protocol implementations [20].

- **Using high level programming languages for distributed applications:** [15] describes a stub compiler for the C++ language. This stub compiler does not need an auxiliary interface definition language. Instead, it uses the operator overloading feature of C++ to enable parameter marshalling. This approach enables distributed applications to be constructed in a straightforward manner. A drawback of using a programming language like C++ is that it allows programmers to use constructs (such as references or pointers) that do not have any meaning on the remote side. Instead, IDLs are more restrictive and disallow such constructs. CORBA IDL has the added advantage that it resembles C++ in many respects and a well-defined mapping from the IDL to C++ has been standardized.

4.3 Application Level Framing and Integrated Layer Processing on Communication Subsystems

Conventional layered protocol stacks lack the flexibility and efficiency required to meet the quality of service requirements of diverse applications running over high speed networks. A remedy for this problem is to use *Application Level Framing* (ALF) [5, 4, 9] and *Integrated Layer Processing* (ILP) [5, 1, 20]. ALF ensures that lower layer protocols deal with data in units specified by the application. ILP provides the implementor with the option of performing all data manipulations in one or two integrated processing loops, rather than manipulating the data sequentially.

5 Concluding Remarks

CORBA is characterized by an integrated approach that supports platform heterogeneity, high system reliability, efficient marshalling and demarshalling of parameters, flexible and efficient object location and selection, and higher level mechanisms for collaboration among services [12]. To meet these requirements and to enhance functionality provided by traditional procedural RPC toolkits (such as Sun RPC and OSF DCE), CORBA supports object-oriented language features. Many sophisticated components must be developed to support features such as remote method invocation, transparent object location and activation, and service selection. These components include directory name servers, object request brokers (ORBs), interface definition language compilers and object locators/traders.

An important class of applications require high-performance communication. Performance-sensitive applications (such as medical imaging or teleconferencing) are not supported efficiently by contemporary CORBA implementations due to presentation layer conversions, data copying, demultiplexing, and memory management overhead. On low-speed networks this overhead is often masked. On high-speed networks, this overhead becomes a significant factor limiting communication performance and ultimately limiting adoption by developers.

In general, the CORBA implementations measured in this paper attain lower throughput than the C, C++ wrapper, and hand-optimized RPC versions of TTCP over ATM. On average, the CORBA performance was around 75-80% the level of the C/C++ versions for remote transfers of scalars and around 33% for `structs` containing binary data. For the loopback tests, the ORBeline version performed as well as the C/C++ versions for scalar data types at higher sender buffer sizes (*e.g.,* for 128 K sender buffer, the ORBeline throughput for sending `doubles` was around 196 Mbps which is comparable to the throughput obtained for the C/C++ versions). The Orbix version did not perform as well as the ORBeline version for transferring scalars (*e.g.,* the Orbix version performed roughly 65-68% as well as the C/C++ versions). Both the CORBA implementations performed poorly compared to the C/C++ versions for transferring `structs` containing binary fields. Both implementations performed roughly 16% as well as the C/C++ versions.

The CORBA implementations performed worst when sending complex typed data (`structs`) because of excessive copying and marshalling/demarshalling overhead and excessive writes resulting from small size write-buffers. The loopback tests provide a means for testing the performance of CORBA and low-level implementations at higher network speeds. From the loopback results, we conclude that with increasing network speeds, the performance of the CORBA implementations becomes worse compared with low-level communication middleware like sockets when marshalling of data is involved. The results in this paper indicate that efficient optimizations need to be applied to the CORBA client-side stubs and server-side skeletons to reduce the marshalling, data copying and request demultiplexing overhead.

We contend that advances in communication middleware like CORBA can be achieved only by simultaneously integrating techniques and tools that simplify application development, optimize application performance, and systematically measure application behavior in order to pinpoint and alleviate performance bottlenecks. Our work is motivated by an increasing demand for efficient and flexible communication software to support next-generation multimedia applications and to leverage emerging high-speed networking technology. We plan to enhance previously described ideas and propose newer schemes for efficient object-to-object communication.

The source code for the various tests performed in this paper is made available through the ACE [22] software distribution.

Acknowledgments

We would like to thank the anonymous reviewers and also Bill Janssen, Karl McCabe and Alan Ewald for their suggestions in improving the paper. We would also like to thank IONA and PostModern Computing for their help in supplying the CORBA implementations used for these tests. Both companies are currently working to eliminate the performance overhead described in this paper. We expect their forthcoming releases to perform much better over high-speed ATM networks.

References

[1] M. Abbott and L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *ACM Transactions on Networking*, 1(5), October 1993.

[2] Kenneth Birman and Robbert van Renesse. RPC Considered Inadequate. In *Reliable Distributed Computing with the Isis Toolkit*, pages 68–78. IEEE Computer Society Press, Los Alamitos, 1994.

[3] G.J Blaine, M.E. Boyd, and S.M. Crider. Project Spectrum: Scalable Bandwidth for the BJC Health System. *HIMSS, Health Care Communications*, pages 71–81, 1994.

[4] Isabelle Chrisment. Impact of ALF on Communication Subsystems Design and Performance. In *First International Workshop on High Performance Protocol Architectures, HIPPARCH '94*, Sophia Antipolis, France, December 1994. INRIA France.

[5] 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.

[6] 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.

[7] 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.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[9] Atanu Ghosh, Jon Crowcroft, Michael Fry, and Mark Handley. Integrated Layer Video Decoding and Application Layer Framed Secure Login: General Lessons from Two or Three Very Different Applications. In *First International Workshop on High Performance Protocol*

Architectures, HIPPARCH '94, Sophia Antipolis, France, December 1994. INRIA France.

[10] 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.

[11] 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.

[12] Object Management Group. *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 edition, March 1994.

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

[14] 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.

[15] Graham Parrington. A Stub Generation System for C++. *Computing Systems*, 8(2):135–170, Spring 1995.

[16] Guru Parulkar, Douglas C. Schmidt, and Jonathan S. Turner. a^{I}t^{P}m: a Strategy for Integrating IP with ATM. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*. ACM, September 1995.

[17] Joseph C. Pasquale, Eric W. Anderson, Kevin R. Fall, and Jonathan S. Kay. High-performance I/O and Networking Software in Sequoia 2000. *Digital Technical Journal*, 7(3), 1995.

[18] Irfan Pyarali, Douglas C. Schmidt, and Timothy H. Harrison. Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging. In *Proceedings of the* 2^{nd} *Conference on Object-Oriented Technologies and Systems*, Toronto, Canada, June 1996. USENIX.

[19] Sanjay Radia, Graham Hamilton, Peter Kessler, and Michael Powell. The Spring Object Model. In *Proceedings of the Conference on Object-Oriented Technologies*, Monterey, CA, June 1995. USENIX.

[20] Antony Richards, Ranil De Silva, Anne Fladenmuller, Aruna Seneviratne, and Michael Fry. The Application of ILP/ALF to Configurable Protocols. In *First International Workshop on High Performance Protocol Architectures, HIPPARCH '94*, Sophia Antipolis, France, December 1994. INRIA France.

[21] Dennis Ritchie. A Stream Input–Output System. *AT&T Bell Labs Technical Journal*, 63(8):311–324, October 1984.

[22] Douglas C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994. USENIX Association.

[23] 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.

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

[25] Sun Microsystems. *Open Network Computing: Transport Independent RPC*, June 1995.

Appendix

The following `struct` declarations are representative of those used for the C/C++ and hand-optimized RPC implementations of TTCP:

C/C++ Struct Declarations

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

typedef struct BinStruct BinStruct;
typedef struct {
        u_long type;
        u_long len;
        double *buffer;
}DoubleSeq;

typedef struct {
        u_long type;
        u_long len;
        BinStruct *buffer;
}StructSeq;
```

RPCL Definition

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

typedef short ShortSeq<>;
typedef long LongSeq<>;
typedef char CharSeq<>;
typedef octet OctetSeq<>;
typedef double DoubleSeq<>;
typedef BinStruct StructSeq<>;

program TTCP {
  version TTCPVERS {
    void SEND_SHORT(ShortSeq) = 1;
    void SEND_LONG(LongSeq) = 2;
    void SEND_CHAR(CharSeq) = 3;
    void SEND_OCTET(OctetSeq) = 4;
    void SEND_DOUBLE(DoubleSeq) = 5;
    void SEND_STRUCT(StructSeq) = 6;
  } = 1;
} = 0x20000001;
```

The following CORBA IDL interface was used for the Orbix and ORBeline CORBA implementations:

```
// sizeof BinStruct == 24 bytes
// due to compiler alignment
struct BinStruct{ short s; char c; long  l;
                   octet o; double d; };

// Richly typed data
interface ttcp_sequence {
  typedef sequence<short> ShortSeq;
  typedef sequence<long> LongSeq;
  typedef sequence<double> DoubleSeq;
  typedef sequence<char> CharSeq;
  typedef sequence<octet> OctetSeq;
  typedef sequence<BinStruct> StructSeq;

  // Routines to send sequences of various data types
  oneway void sendShortSeq (in ShortSeq ttcp_seq);
  oneway void sendLongSeq (in LongSeq ttcp_seq);
  oneway void sendDoubleSeq (in DoubleSeq ttcp_seq);
  oneway void sendCharSeq (in CharSeq ttcp_seq);
  oneway void sendOctetSeq (in OctetSeq ttcp_seq);
  oneway void sendStructSeq (in StructSeq ttcp_seq);

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