Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks

Aniruddha S. Gokhale and Douglas C. Schmidt
{gokhale,schmidt}@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130[1]

The paper appeared in the Proceedings of ICDCS '97, May 27-30, 1997 in Baltimore, Maryland.

Abstract

Conventional implementations of CORBA *communication middleware incur significant overhead when used for performance-sensitive applications over high-speed networks. As gigabit networks become pervasive, inefficient middleware will force programmers to continue using lower-level mechanisms to achieve necessary transfer rates and end-to-end latency. This is a serious problem for mission/life-critical applications (such as real-time avionics, process control systems, and medical imaging).*

This paper provides two contributions to the study of CORBA *performance over high-speed networks. First, we measure the latency of various types and sizes of oneway and twoway client requests using a pair of widely used implementations of two C++ implementations of* CORBA *– Orbix 2.1 and VisiBroker 2.0. Second, we use Orbix and VisiBroker to measure the scalability of* CORBA *servers in terms of the number of objects they can support efficiently. These experiments extend our previous work on* CORBA *performance for bandwidth-sensitive applications (such as satellite surveillance, medical imaging, and teleconferencing).*

Our results show that the latency for CORBA *implementations is relatively high and server scalability is relatively low. Our latency experiments show that non-optimized internal buffering and presentation layer conversion overhead in* CORBA *implementations can cause substantial delay variance, which is unacceptable in many real-time or constrained-latency applications. Likewise, our scalability experiments reveal that neither Orbix nor VisiBroker can handle a large number of objects in a single server process. The paper concludes by outlining optimizations we are developing to overcome the performance limitations with existing* CORBA *implementations.*

1 Introduction

Applications and services for next-generation distributed systems must be reliable, flexible, reusable, and capable of providing scalable, low-latency quality of service to delay-sensitive applications. In addition, communication software must allow bandwidth-sensitive applications to realize high-speed data transfers over gigabit networks. Reliability, flexibility, and reusability are essential to respond rapidly to changing application requirements that span a wide range of media types and access patterns [1].

These requirements motivate the use of the *Common Object Request Broker Architecture* (CORBA) [2]. CORBA is designed to enhance distributed applications by automating common networking tasks such as parameter marshaling, object location and object activation, as well as providing the basis for defining higher layer distributed services (such as naming, events, replication, and transactions) [3].

The success of CORBA in mission-critical distributed computing environments depends heavily on the ability of Object Request Brokers (ORBs) to provide:

- **High bandwidth** CORBA implementations must provide high throughput to bandwidth-sensitive applications (such as medical imaging, satellite surveillance, or teleconferencing systems);

- **Low latency** CORBA implementations must support low latency for delay-sensitive applications (such as real-time avionics, distributed interactive simulations, and telecommunication systems);

- **Scalability of endsystems and distributed systems** CORBA implementations must scale effectively as the number of objects in endsystems and distributed systems increases. Scalability is important for large-scale applications (such as enterprise-wide network management systems), which must handle a large number of objects on each network node, as well as a large number of nodes throughout a distributed computing environment.

[1] This work was supported in part by NSF grant NCR-9628218.

High-speed networks (such as ATM and FDDI) now support quality of service guarantees in terms of bandwidth and latency. However, as shown in Section 4, conventional implementations of CORBA incur significant overhead when used for latency-sensitive applications over high-speed networks. If not corrected, this overhead will force developers to avoid CORBA middleware and continue to use lower-level tools (such as sockets). Unfortunately, lower-level tools fail to provide other key benefits of CORBA such as reliability, flexibility and reusability, which are crucial to the success of complex constrained-latency distributed applications [4]. Therefore, it is imperative to eliminate the sources of CORBA overhead shown in this paper.

Previous work [5, 6, 7] has focused on the *throughput performance* of CORBA implementations that transfer large amounts of untyped and richly-typed data. This paper extends earlier work by focusing on *latency* and *scalability*. The research contributions of this paper include the following:

• CORBA **Latency** We measure the performance of two widely used implementations of CORBA (Orbix 2.1 and VisiBroker 2.0) to determine their oneway and twoway latencies for sending various types (such as `chars`, `shorts`, `longs`, and `structs`) and sizes (such as 1, 2, 4, ..., 1,024 units of each data type) of client requests using various invocation strategies (such as static invocation and dynamic invocation). The results of our latency experiments are described in Section 4.

• CORBA **Scalability** We measure how the number of objects in a server affect an ORB's ability to process client requests efficiently. This paper focuses on the scalability of CORBA implementations in endsystem servers (*i.e.,* the number of objects in a server process), rather than distributed scalability (*i.e.,* the number of endsystems in a network). The results of our endsystem scalability experiments are described in Section 4.

In the paper, we pinpoint precisely where the key sources of latency and scalability overhead exist in conventional implementations of CORBA. We used two widely available CORBA implementations (Orbix 2.1 and VisiBroker 2.0) for our experiments. Our findings indicate that the latency overhead of CORBA implementations stem from a variety of sources including (1) lack of integration with advanced OS and network features, (2) inefficient server demultiplexing techniques, (3) long chains of intra-ORB function calls, (4) excessive presentation layer conversions and data copying, and (5) non-optimized buffering algorithms used for network reads and writes.

The paper is organized as follows: Section 2 outlines the CORBA communication middleware architecture; Section 3 describes our CORBA/ATM testbed and experimental methods; Section 4 presents the key sources of latency and scalability overhead in conventional CORBA implementations over ATM; Section 5 describes our research on developing optimizations that eliminate the performance bottlenecks in existing CORBA implementations; Section 6 describes related work; and Section 7 presents concluding remarks.

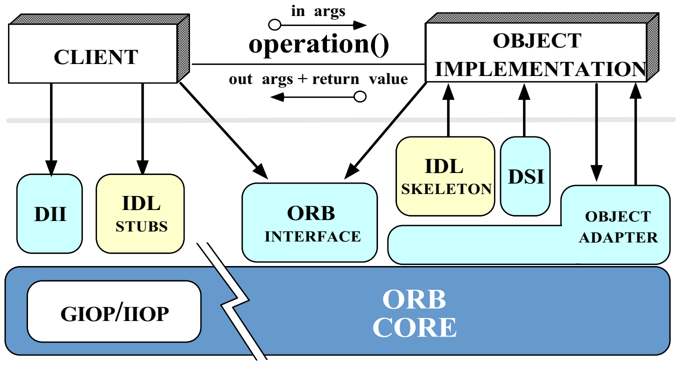

Figure 1: Components in the CORBA Distributed Object Computing Model

2 Overview of the CORBA Architecture

CORBA is an open standard for distributed object computing [2]. Figure 1 illustrates the primary components in the CORBA architecture. The CORBA standard defines a set of components that allow client applications to invoke operations (`op`) with arguments (`args`) on object implementations. CORBA enhances application flexibility since object implementations can be configured to run locally and/or remotely without affecting their implementation or use. The responsibility of each component in CORBA and its implications for high-performance distributed computing systems 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 the response (if any) 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 functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface (DII) 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. Stubs marshal typed data objects from a high-level application representation to a low-level packet representation, whereas skeletons demarshal the low-level packet representation back into a typed data object that is meaningful to the application. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler. The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.

• **Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI)** The DII allows a client to 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 the process. Unlike IDL stubs (which only allow RPC-style twoway and oneway requests), the DII also allows clients to make non-blocking *deferred synchronous* calls, which separate send and receive operations.

The DSI is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation or ORB bridge that has no compile-time knowledge of the type of object it is implementing. The client making the request need not be aware that the implementation is using the type-specific IDL skeletons or the dynamic skeletons.

• **Object Adapter** The Object Adapter assists the ORB by demultiplexing requests to the target object and dispatching operation upcalls on the object. In addition, the 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, library Object Adapters for non-remote objects, and real-time Object Adapters [8] for applications that require QoS guarantees).

The use of CORBA as communication middleware enhances application flexibility and portability by automating many common development tasks such as object location, parameter marshaling, 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 more flexible communication mechanisms (such as object references that support peer-to-peer communication and dynamic invocation capabilities). These features enable complex distributed and concurrent applications to be developed more rapidly and correctly.

The primary drawback to using higher-level middleware like CORBA is its potential for low throughput, high latency, and lack of scalability over high-speed networks. In general, existing implementations of CORBA have not been optimized

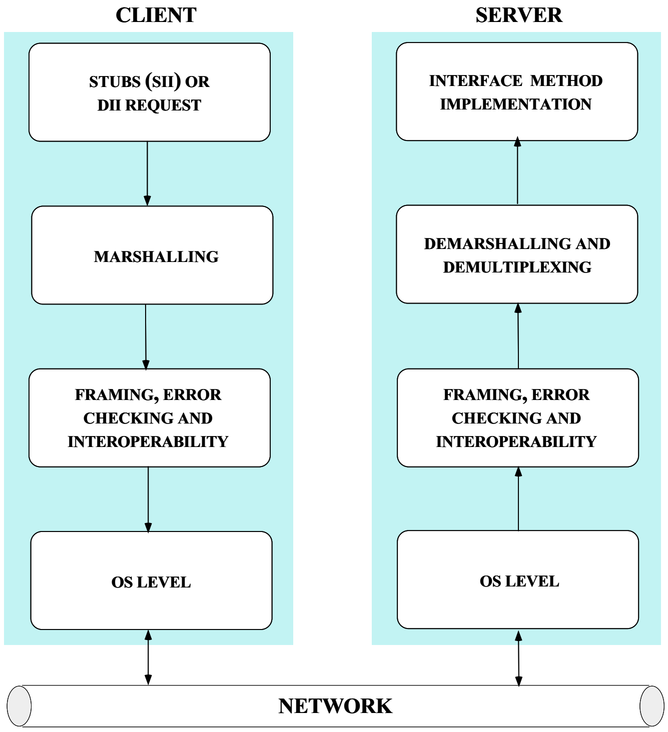

Figure 2: General Path of CORBA Requests

significantly since performance has not generally been an issue on low-speed networks. Figure 2 shows the general path that CORBA implementations use to transmit requests from client to server for remote operation invocations. In Section 4.3, we show the corresponding path for the Orbix 2.1 and VisiBroker 2.0 ORBs and precisely pinpoint the sources of overhead existing along the data path. It is beyond the scope of this paper to discuss the limitations with the CORBA communication model (see [9] for a synopsis).

3 CORBA/ATM Testbed and Experimental Methods

This section describes our CORBA/ATM testbed and outlines our experimental methods.

3.1 Hardware and Software Platforms

The experiments in this section were conducted using a FORE systems ASX-1000 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.5.1. The ASX-1000 is a 96 Port, OC12 622 Mbs/port switch. Each UltraSparc-2 contains two 168 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.5.1 TCP/IP protocol stack is implemented using the STREAMS communication framework. Each UltraSparc-2 has 256 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.

3.2 Traffic Generators

Our earlier studies [5, 6, 4, 7] tested bulk data performance using "flooding models" that transferred untyped bytestream data, as well as richly typed data between hosts using several implementations of CORBA and other lower-level mechanisms like sockets. On the client-side, these experiments measured the static invocation interface (SII) and the dynamic invocation interface (DII) provided by the CORBA implementations. The SII allows a client to invoke a remote method via static stubs generated by a CORBA IDL compiler, which is useful when client applications know the interface offered by the server at compile-time. In contrast, the DII allows a client to access the underlying request mechanisms provided by an ORB directly, which is useful when the applications do not know the interface offered by the server until run-time.

The experiments conducted for this paper extend our earlier throughput studies by measuring the end-to-end latency and request demultiplexing overhead that is incurred when invoking various request sizes on a range of objects maintained by a CORBA server. In addition, we measure CORBA scalability by determining the performance impact from increasing the number of objects in an endsystem server process.

Traffic for the latency experiment was generated and consumed by the Orbix 2.1 and VisiBroker 2.0 implementations of TTCP [10] (TTCP is a widely used benchmarking tool to evaluate the performance of TCP/IP and UDP/IP networks). In this case, we defined both oneway and twoway CORBA operations. The flow of control is uni-directional in oneway operations (*i.e.,* the client need not block until the server executes the operation), whereas in twoway operations the client blocks until the operation returns. We measured round-trip latency using the twoway CORBA operations. Each operation transfers a `sequence` of various data types. In addition, we measured operations that did not use any parameters to determine "best case" latency.

The following data types were used for all of the tests: primitive types (`short`, `char`, `long`, `octet`, `double`) and a C++ `struct` composed of all the primitives (`BinStruct`). The CORBA implementation transferred the data types using IDL `sequences`, which are dynamically-sized arrays. The IDL definition used in the test is shown in the Appendix A.

3.3 TTCP Parameter Settings

Earlier studies [11, 12, 13, 4, 7] of transport protocol performance over ATM demonstrate the performance impact of parameters such as the size of socket queues, data buffer, and number of target objects on an endsystem (*e.g.,* a server). Therefore, our TTCP benchmarks systematically varied these parameters for each type of data as follows:

- **Socket queue size** The sender and receiver socket queue sizes used were 64 K bytes, which is the maximum on SunOS 5.5. These parameters influence the size of the TCP segment window, which has been shown [13, 7] to significantly affect CORBA-level and TCP-level performance on high-speed networks.

- **TCP "No Delay" option** Since the request sizes for our tests are relatively small, the `TCP_NODELAY` option is set on the client side. Without the `TCP_NODELAY` option, the client's TCP uses Nagel's algorithm, which buffers "small" requests until the preceding small request is acknowledged. On high-speed networks the use of Nagel's algorithm can increase latency unnecessarily. Since this paper focuses on measuring the latency of small requests, we enabled the `TCP_NODELAY` option to send packets immediately.

- **Data buffer size** For the latency measurements, the sender transmits parameter units of a specific data type incremented in powers of two, ranging from 1 to 1,024. Thus, for `shorts` (which are two bytes long on the SPARCs), the sender buffers ranged from 2 bytes to 2,048 bytes. In addition, we also measured the latency of remote method invocations that had no parameters.

- **Number of target objects** Increasing the number of objects on the server increases the demultiplexing effort required to dispatch the incoming request to the appropriate object. To pinpoint the latency overhead in this demultiplexing process, and to evaluate the scalability of CORBA implementations, our experiments used a range of objects (1, 100, 200, 300, 400, and 500) on the server.

3.4 Profiling Tools

Detailed timing measurements used to compute latency were made with the `gethrtime` system call available on SunOS 5.5. This system call uses the SunOS 5.5 high-resolution timer, which expresses time in nanoseconds from an arbitrary time in the past. The time returned by `gethrtime` is very accurate since it does not drift.

The profile information for the empirical analysis was obtained using the `Quantify` performance measurement tool. `Quantify` analyzes performance bottlenecks and identifies sections of code that dominate execution time. Unlike traditional sampling-based profilers (such as the UNIX `gprof` tool), `Quantify` reports results without including its own overhead. In addition, `Quantify` measures the overhead of system calls and third-party libraries without requiring access to source code.

3.5 Invocation Strategies

One source of latency incurred by CORBA implementations in high-speed networks involves the *operation invocation* *strategy*. This strategy determines whether the requests are invoked via the static or dynamic interfaces and whether the client expects a response from the server. In our experiments, we measured the following operation invocation strategies defined by the CORBA specification:

• **Oneway static invocation** The client uses the SII stubs generated by the CORBA IDL compiler for the oneway operations defined in the IDL interface. For oneway operations the CORBA standard specifies "best-effort" delivery semantics, *i.e.,* no application-level acknowledgment is passed back to the requester. IONA and Visigenic both use TCP/IP to transmit `oneway` operations;

• **Twoway static invocation** The client uses the static invocation interface (SII) stubs for `twoway` operations defined in IDL interfaces. After each `twoway` operation is invoked the client blocks until an acknowledgment is received (*i.e.,* the operation returns). In our experiments, we defined `twoway` operations to return "`void`," thereby minimizing the size of the acknowledgment from the server;

• **Oneway dynamic invocation** The client uses the DII to build a request at run-time and uses the CORBA `Request` class to make the requests;

• **Twoway dynamic invocation** The client uses the dynamic invocation interface (DII) to make the requests, but blocks until the call returns from the server.

We measured the average latency for 100 client requests for every 1, 100, 200, 300, 400, and 500 objects on the server. In every request, we invoked the same method. We restricted the number of requests per object to 100 since neither ORB could handle a larger numbers of requests without crashing.

3.6 Target Object Demultiplexing Strategies

Another source of overhead incurred by CORBA implementations in high-speed networks involves the time the ORB's Object Adapter spends demultiplexing requests to target objects. The type of demultiplexing strategy used by an ORB significantly affects its scalability. Scalability is important for applications like enterprise-wide network management systems, which must handle agents containing a potentially large number of managed objects on each ORB endsystem.

Conventional ORBs (including Orbix and VisiBroker) demultiplex client requests to the appropriate operation of the target object implementation using the following steps (shown in Figure 3):

• **Steps 1 and 2** The OS protocol stack demultiplexes the incoming client request multiple times (*e.g.,* through the data link, network, and transport layers, as well as the user/kernel boundary) to the ORB's Object Adapter;

• **Steps 3 and 4** The Object Adapter uses the addressing information in the client request to locate the appropriate target object implementation and associated IDL skeleton;

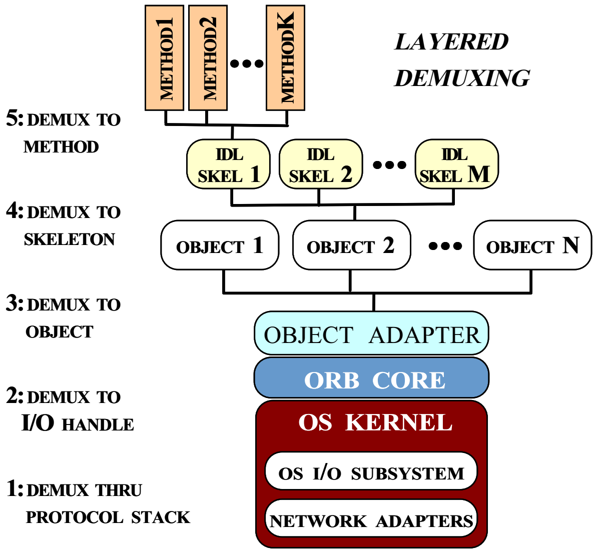

Figure 3: Demultiplexing Client Requests to CORBA Target Object Implementations

• **Step 5** The IDL skeleton locates the appropriate operation, demarshals the request buffer into the operation parameters, and performs the upcall to the operation.

Demultiplexing client requests through all these layers is expensive, particularly when a large number of operations appear in an IDL interface and/or a large number of objects are managed by an ORB.

Our prior work [5, 14] analyzed the impact of various IDL skeleton demultiplexing techniques (such as linear search and direct demultiplexing). However, in many applications the number of operations defined per-IDL interface is relatively small and static, compared to the number of potential objects, which can be quite large and dynamic. To evaluate the scalability of the CORBA implementations in this paper, therefore, we varied the number of objects residing in the server process from 1 to 500, by increments of 100. The server used the *shared* activation mode, where all objects on the server are managed by the same process.

3.7 Request Generation Algorithms

One way to optimize the demultiplexing overhead described above is to have the Object Adapter cache recently accessed target objects. Caching is particularly useful if client operations arrive in "request trains," where a server receives a series of requests for the same target object. By caching information about an object, the server can reduce the overhead of locating the object for every incoming request.

The CORBA standard does not mandate the use of caching. Therefore, ORB implementations are not obliged to support it. To determine if caching was used (and to measure its effectiveness), we devised the following two algorithms, *Request Train* and *Round Robin*, to invoke client requests and calculate average latency:

• **Request Train algorithm** The Request Train algorithm generates client requests in a manner that measures the presence and benefits of ORB Object Adapter caching, as follows:

```
const int MAXITER = 100;

long sum = 0;
Profile_Timer timer; // Begin timing.

for (int j = 0; j < num_objects; j++){
  for (int i = 0; i < MAXITER; i++) {
     // Use one of the 2 invocation strategies
     // to call the send() method on object_#j
     // at the server...
     sum += timer.current_time ();
  }
}
avg_latency = sum / (MAXITER * num_objects);
```

This algorithm does not change the destination object until `MAXITER` requests are performed. If a server is caching information about recently accessed objects, the Request Train algorithm should elicit different performance characteristics than the Round Robin algorithm described next.

• **Round Robin algorithm** The Round Robin algorithm iterates `MAXITER` times, each time invoking the `send` operation on a different object reference (an object reference is a client-side entity that behaves as a proxy on behalf of the object residing on the server). The Round Robin algorithm is defined as follows:

```
const int MAXITER = 100;

long sum = 0;
Profile_Timer timer; // Begin timing.

for (int i = 0; i < MAXITER; i++){
  for (int j = 0; j < num_objects; j++) {
     // Use one of the 2 invocation strategies
     // to call the send() method on object_#j
     // at the server...
     sum += timer.current_time ();
  }
}
avg_latency = sum / (MAXITER * num_objects);
```

4 Performance Results for CORBA Latency and Scalability over ATM

This section presents the performance results from our latency and scalability experiments. Sections 4.1 and 4.2 describe the blackbox experiments that measure end-to-end communication delay from client requester to a range of server target objects using a variety of types and sizes of data. Section 4.3 describes a whitebox empirical analysis using `Quantify` to precisely pinpoint the overheads that yield these results. Our measurements include the overhead imposed by all the layers shown in Figure 2.

4.1 Blackbox Results for Parameterless Operations

In this section, we describe the results for sending parameterless operations using the Round Robin and Request Train invocation strategies.

Latency and scalability of parameterless operations Figures 4 and 5 depict the average latency for the parameterless operations using the Request Train variation of our request generation algorithm. Likewise, Figures 6 and 7 depict the average latency for invoking parameterless operations using the Round Robin algorithm.

These figures reveal that the results for the Request Train experiment and the Round-Robin experiment are essentially identical. Thus, it appears that neither ORB supports caching of server objects. As a result, the remainder of our tests only use the Round Robin algorithm.

- *Twoway latency* – The figures illustrate that that the performance of VisiBroker was relatively constant for twoway latency. In contrast, Orbix's latency grew as the number of objects increased. The rate of increase was approximately 1.12 times for every 100 additional objects on the server.

  Tracing the CORBA run-time system calls using `truss` revealed the problem with Orbix. When Orbix 2.1 is run over ATM networks, it opens a new TCP connection (and thus a new socket descriptor) for every object reference.[2] This has the following consequences:

  – *Increased demultiplexing overhead* – Opening a new socket connection for each object reference degrades latency significantly since the OS kernel must search the socket endpoint table to determine which descriptor should receive the data.
  – *Limited scalability* – As the number of objects grows, all the available descriptors on the client and server were exhausted. We used the UNIX `ulimit` command to increase the number of descriptors to 1,024, which is the maximum supported per-process on SunOS 5.5 without reconfiguring the kernel. Thus, we were limited to approximately 1,000 object references per-server process on Orbix over ATM.

  In contrast, VisiBroker did not create socket descriptors for every object reference. Instead, a single connection and socket descriptor were shared by all object references in the client. Likewise, a single connection and socket descriptor were shared by all target object implementations in the server. This, combined with its hashing-based demultiplexing scheme for locating objects and methods, significantly reduces latency. In addition, we were able to obtain object references for more than 1,000 objects.

- *Oneway latency* – The figures illustrate that in case of VisiBroker, the oneway latency remains roughly constant as the number of objects on the server increase. However, Orbix's latency grows as the number of objects increase. Figures 4 and 6 reveal an interesting

[2] Interestingly, when the Orbix client is run over Ethernet it only uses a single socket on the client, regardless of the number of objects in the server process.

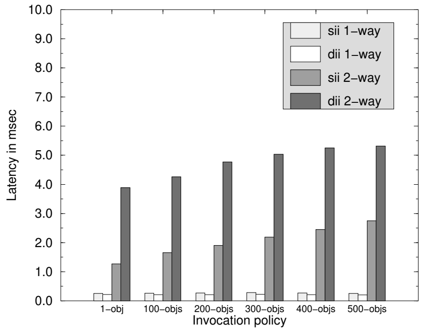

Figure 4: Orbix: Latency for Sending Parameterless Operation using Request Train Requests

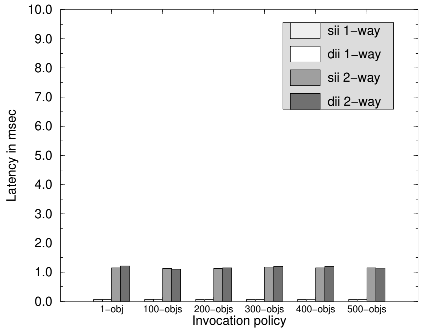

Figure 5: VisiBroker: Latency for Sending Parameterless Operation using Request Train Requests

case with Orbix's oneway latency. The oneway SII and DII latencies remain slightly less than their corresponding twoway latencies until 200 objects on the server. Beyond this, the oneway latencies exceed their corresponding twoway latencies. Orbix opens a new TCP connection (and hence a new socket descriptor) for every object reference. Since the oneway calls do not involve any server response to the client, the client is able to send requests without blocking. As explained in Section 4.3.3, due to the large number of open TCP connections and inefficient demultiplexing strategies, the receiver is unable to keep pace with the sender. As a result the underlying transport protocol (TCP in this case) is required to invoke flow control techniques to slow down the sender. As the number of objects increase, this flow control overhead becomes dominant thereby increasing oneway latency. In contrast, the twoway latency does not incur this flow control overhead since the sender blocks for a response after every request.

Figure 8 compares the twoway latencies obtained for sending parameterless operations for Orbix and VisiBroker with that of a low-level C implementation that uses sockets. The twoway latency comparison reveals that the VisiBroker and Orbix versions perform only 50% and 46% as well as the C

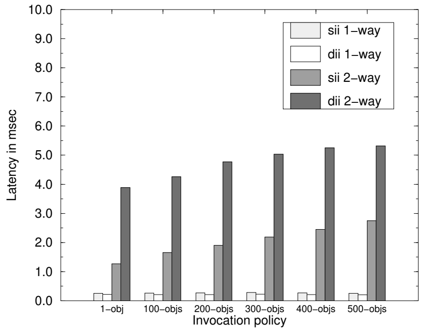

Figure 6: Orbix: Latency for Sending Parameterless Operation using Round Robin Requests

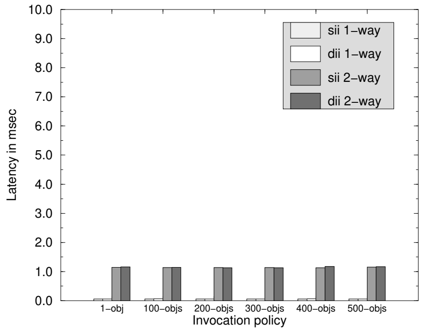

Figure 7: VisiBroker: Latency for Sending Parameterless Operation using Round Robin Requests

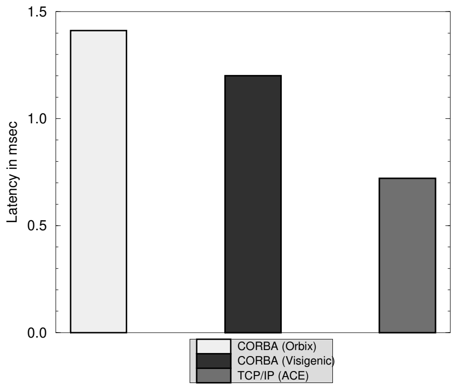

Figure 8: Comparison of Twoway Latencies

version, respectively.

4.1.1 Summary of Results for Parameterless Operations

- Neither ORB we measured caches recently accessed object references in the Object Adapter. As a result, the latency for the Request Train and Round Robin cases are nearly equivalent;
- Oneway latencies for VisiBroker remained relatively constant with increase in the number of objects on the server. However, the oneway latency for Orbix increases roughly linearly with increase in the number of objects;
- Oneway latencies for Orbix exceed their corresponding twoway latencies beyond 200 objects on the server. This is due primarily to the flow control mechanism used by the underlying transport protocol to slow down the fast sender;
- Twoway latency for VisiBroker remains relatively constant as the number of objects increases. This is due primarily to the efficient demultiplexing based on hashing used by VisiBroker. In addition, VisiBroker does not open a new connection for every object reference;
- Twoway latency for Orbix increases linearly at a rate of roughly 1.12 per 100 object increment. As explained earlier, this is due primarily to the inefficient demultiplexing strategy and an open TCP connection per object reference;
- Twoway DII latency in VisiBroker is comparable to its twoway SII latency. This is due to the ability to reuse CORBA thereby requiring to create the request only once. The CORBA 2.0 specification does not dictate which of these semantics is correct, so an implementation is free to use either approach.
- Twoway DII latency in Orbix is roughly 2.6 times that of its twoway SII latency. In Orbix DII, a new request has to be created per invocation;
- Twoway DII latency for Orbix is always greater than its twoway SII latency, whereas for VisiBroker they are comparable. The reasons is that Orbix creates a new request for every DII invocation. In contrast, VisiBroker recycles the request.

4.2 Blackbox Results for Parameter Passing Operations

Figures 9 through 16 depict the average latency for sending richly-typed `struct` data and untyped `octet` data using (1) the twoway operation invocation strategies (described in Section 3.5) and (2) varying the number of server-side objects (described in Section 3.6). These figures reveal that as the sender buffer size increases the marshaling and data copying overhead also grows [5, 6], thereby increasing latency. These results demonstrate the benefit of using more efficient buffer management techniques and highly optimized stubs to reduce the presentation conversion and data copying overhead.

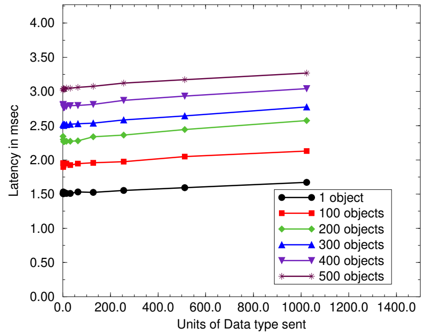

Figure 9: Orbix Latency for Sending Octets Using Twoway SII

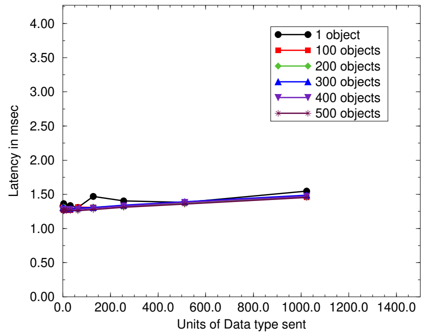

Figure 10: VisiBroker Latency for Sending Octets Using Twoway SII

- *Twoway latency* – Figures 9 through 16 reveal that the twoway latency for Orbix increases as (1) the number of server objects and (2) the sender buffer sizes increase. In contrast, for VisiBroker the latency increases only with the size of sender buffers. Figures 13 through 16 also reveal that the latency for the Orbix twoway SII case at 1,024 data units of `BinStruct` is almost 1.2 times that for VisiBroker.

  Similarly, the latency for the Orbix twoway DII case at 1,024 data units of `BinStruct` is almost 4.5 times that for VisiBroker. In addition, the figures reveal that for Orbix, the latency increases as the number of server objects increase. As shown in 4.3, this is due primarily to the inefficient demultiplexing strategy used by Orbix. For VisiBroker, the latency remains unaffected as the number of objects increases.

  Orbix incurs higher latencies than VisiBroker due to (1) the additional overhead stemming from the inability of Orbix DII to reuse requests and (2) the presentation layer overhead of marshaling and demarshaling the

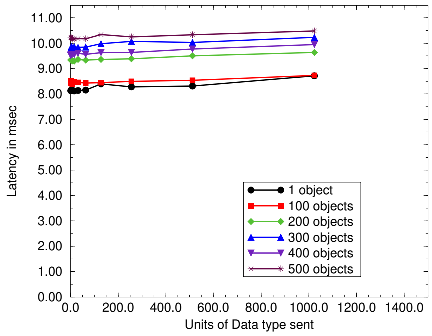

Figure 11: Orbix Latency for Sending Octets Using Twoway DII

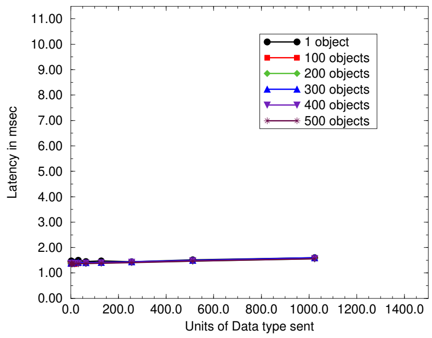

Figure 12: VisiBroker Latency for Sending Octets Using Twoway DII

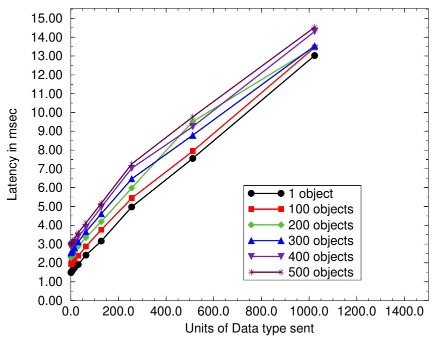

Figure 13: Orbix Latency for Sending Structs Using Twoway SII

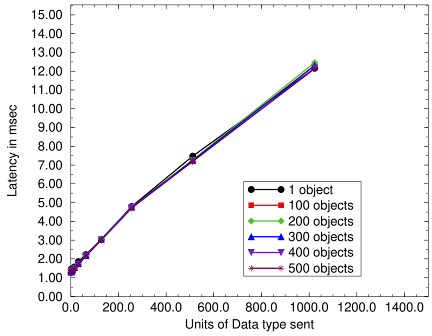

Figure 14: VisiBroker Latency for Sending Structs Using Twoway SII

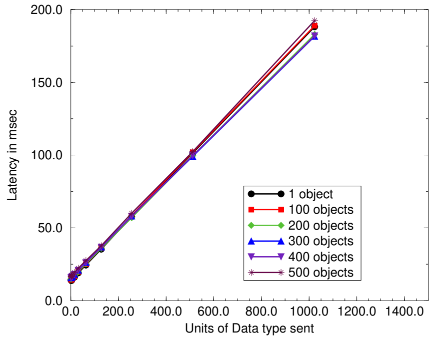

Figure 15: Orbix Latency for Sending Structs Using Twoway DII

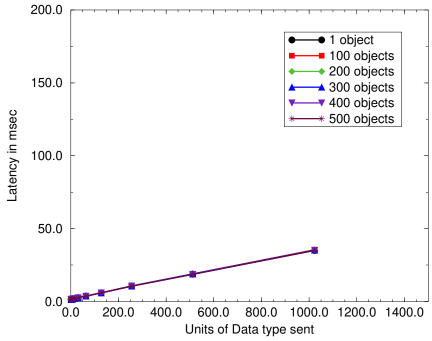

Figure 16: VisiBroker Latency for Sending Structs Using Twoway DII

`BinStructs`. These sources of overhead reduce the receiver's performance, thereby triggering the flow control mechanisms of the transport protocol, which impede the sender's progress.

[5, 6] precisely pinpoint the marshaling and data copying overheads when transferring richly-typed data using SII and DII. The latency for sending `octets` is significantly less than that for `BinStructs` due to significantly lower overhead of presentation layer conversions. Section 4.3 presents our analysis of the `Quantify` results for sources of overhead that increase the latency of client request processing.

4.2.1 Summary of Latency and Scalability Results for CORBA Parameter Passing Operations

The following summarizes the latency results for parameter passing operations described above:

- Latency for both Orbix and VisiBroker increases linearly with the size of the request. This is due to the increased parameter marshaling overhead;
- VisiBroker exhibits relatively low, constant latency as the number of objects increase, due to its use of hashing-based demultiplexing for objects and IDL skeletons at the receiver. In contrast, Orbix exhibits linear increases in latency based on the number of server objects and the number of operations in an IDL interface. This behavior stems from Orbix's use of linear search at the TCP layer since it opens a connection per object reference. In addition, it also uses linear search for `string` comparisons in its IDL skeletons;
- The DII performs consistently worse than SII (For twoway Orbix – 3 times for `octets`, 14 times for `BinStructs`; for VisiBroker – comparable for `octets`, and roughly 4 times for `BinStructs`). Since Orbix does not support reusing the requests, the DII latency for Orbix incurs additional overhead. However, both Orbix and VisiBroker have to populate the request with parameters. This involves marshaling and demarshaling the parameters. The marshaling overhead for `BinStructs` is more significant than that for `octets`. As a result, the DII latency for `BinStructs` is worse compared to that of `octets`.

4.3 Whitebox Analysis of Latency and Scalability Overhead

Sections 4.1 and 4.2 presented the results of our blackbox performance experiments. This section presents the results of our whitebox profiling to illustrate *why* the two ORBs performed differently. We analyze the `Quantify` reports on the sources of latency and scalability overhead in CORBA implementations to help explain the variation in the performance results reported in Section 4.

Figures 17 and 18 show the path a CORBA request takes through the client and server using the SII stubs generated

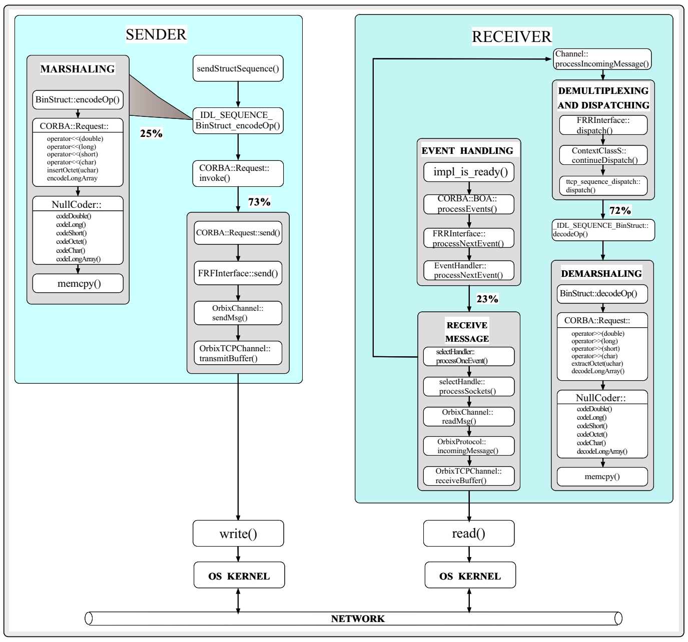

Figure 17: Request Path Through Orbix Sender and Receiver for SII

by Orbix and VisiBroker IDL compilers, respectively. In addition, the figures show how these two ORBs implement the generic request path shown in Figure 2. Percentages at the side of each figure indicate the contribution to the total processing time for a call to the `sendStructSeq` method, which was used to perform the operation for sending sequences of `BinStructs`. The percentages in the figures do not add up to 100 since we do not show the entire contribution of the OS and network device overhead.

The DII request path is similar to the SII path, except that the clients create the request at run-time rather than using the SII stubs generated by the IDL compiler.

4.3.1 Orbix SII Request Flow Overhead

In Figure 17, the Orbix sender invokes the stub for the `ttcp_sequence::sendStructSeq` method. The request traverses through the `invoke` and the `send` routines of the CORBA `Request` class and ends up in the `OrbixChannel` class. At this point, it is handled by a specialized class, `OrbixTCPChannel`, which uses the TCP/IP protocol for communication.

On the receiver side, the request travels through a series of dispatcher classes that locate the intended target object implementation and its associated IDL skeleton. Finally, the `ttcp_sequence_dispatch` class demultiplexes the incoming request to the appropriate target object implementation and dispatches its `sendStructSeq` method with the demarshaled parameters.

On the sender-side, most of the overhead is attributed to the operating system. The Orbix version uses the `write`

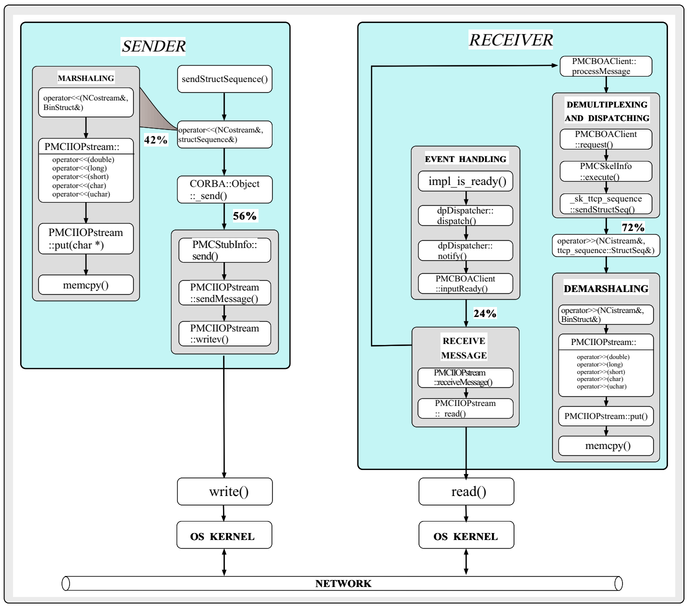

Figure 18: Request Path Through VisiBroker Sender and Receiver for SII

system call which accounts for 73% of the processing time, due primarily to TCP/IP protocol processing in the SunOS kernel. The rest of the overhead is attributable to marshaling and data copying, which accounts for roughly 25% of the processing time. On the receiver side, the demarshaling layer accounts for almost 72% of the overhead, due largely to the presentation layer conversion overhead incurred while demarshaling incoming parameters.

4.3.2 VisiBroker SII Request Flow Overhead

In Figure 18, the VisiBroker sender invokes the stub for the `sendStructSeq` method defined by the `ttcp_sequence` class. The request passes through the `send` methods of the `CORBA::Object` and the `PMCStubInfo` classes. Finally, the request passes through the methods of the `PMCIIOPStream` class, which implements the Internet Inter-ORB Protocol (IIOP)[2][3] The IIOP implementation writes to the underlying socket descriptor.

On the receiver side, the IIOP implementation reads the packet using methods of the `PMCIIOPStream` class, which passes the request to the basic object adapter (BOA). The BOA demultiplexes the incoming request by identifying the skeleton (`_sk_ttcp_sequence::skeleton`). The skeleton identifies the object implementation and makes an upcall to the `sendStructSeq` method of the `ttcp_sequence_i` implementation class.

On the sender-side, 56% of the overhead is attributed to the operating system and networking level. The rest of the overhead is attributable to marshaling and data copying which accounts for roughly 42% of the processing time. On the receiver side, the demarshaling and demultiplexing layer accounts for almost 72% of the processing time. The VisiBroker implementation spends most of the receiver-side processing time in demarshaling the parameters. In addition, the incoming parameters have to travel through long chain of function calls (shown in Figure 18), which increase the overhead.

[3] The IIOP specifies a standard communication protocol between objects on different nodes or between heterogeneous ORBs.

Comm. Entity	Request Train	Analysis: Method Name	msec	%
`Client`	`No`	`read`	112,795	99
	`Yes`	`read`	64,075	99
`Server`	`No`	`strcmp`	2,559	21.79
		`hashTable::lookup`	1,852	15.77
		`write`	949	8.08
		`select`	768	6.54
		`hashTable::hash`	600	5.10
		`Selecthandler::processSockets`	406	3.45
		`read`	346	2.95
	`Yes`	`strcmp`	2,468	21.35
		`hashTable::lookup`	1,810	15.66
		`write`	960	8.30
		`select`	761	6.59
		`hashTable::hash`	595	5.14
		`Selecthandler::processSockets`	404	3.50
		`read`	323	2.79

Table 1: Analysis of Target Object Demultiplexing Overhead for Orbix

4.3.3 Target Object Demultiplexing Overhead

To evaluate how the CORBA implementations scale for endsystem servers, we instantiated 1, 100, . . . , 500 objects on the server. The following discussion analyzes the server-side overhead for demultiplexing client requests to target objects. We analyze the performance of the `sendNoParams_1way` method for 500 objects on the server and 10 iterations. The `sendNoParams_1way` method is chosen so that the demultiplexing overhead can be analyzed without being affected by the demarshaling overhead involved with sending richly-typed data as shown in Sections 4.3.1 and 4.3.2.

• **Orbix demultiplexing overhead** Table 1 depicts the affect on latency and scalability of instantiating 500 objects and invoking 10 requests of the `sendNoParams_1way` method per object using Orbix. `Quantify` analysis reveals that the performance of both the Round Robin and the Request Train case is similar. In both cases, the client spends most of its time performing network reads.

The server spends ∼22% of its time doing `strcmps` used for linearly searching the operation table to lookup the right operation, 16% of the time doing hash table lookups to lookup the right object and its skeleton, ∼8% of the time in `writes`, and ∼7% of its time in `select`. Orbix opens a new socket

Comm. Entity	Request Train	Analysis		
		Method Name	msec	%
Client	No	write	10,895	99.00
	Yes	write	10,992	99.00
Server	No	write	393	20.84
		~NCTransDict	138	7.31
		~NCClassInfoDict	138	7.31
		read	83	4.40
		NCOutTbl	73	3.84
		NCClassInfoDict	71	3.75
	Yes	write	275	15.32
		~NCTransDict	138	7.67
		~NCClassInfoDict	138	7.67
		read	83	4.61
		NCOutTbl	73	4.03
		NCClassInfoDict	71	3.93

Table 2: Analysis of Target Object Demultiplexing Overhead for VisiBroker

descriptor for every object reference obtained by the client. Hence, to demultiplex incoming requests, Orbix must use `select` to determine which socket descriptor is ready for reading.

• **VisiBroker demultiplexing overhead** Table 2 depicts the affect on latency and scalability of instantiating 500 objects and invoking 10 iterations of the `sendNoParams_1way` method per object using VisiBroker. The table revels no significant difference between the Round Robin and Request Train case. The `Quantify` analysis for the VisiBroker version reveals that the server spends ~15-20% of its time in network writes, ~5% in `reads`, and ~22% time demultiplexing requests. The demultiplexing process involves reading and managing the internal tables (`~NCTransDict`, `NCOutTbl`). These internal tables are used by VisiBroker's hash-based table lookup strategy to demultiplex and dispatch the incoming request to the intended object.

Comparison of Orbix and VisiBroker demultiplexing overhead

- VisiBroker opens one socket descriptor for all object references in the same server process. In contrast, Orbix opens a new socket descriptor for every object reference over ATM networks (described in Section 4.1);
- VisiBroker uses a hashing-based scheme to demultiplex incoming requests to their target object. In contrast, although Orbix also uses hashing to identify the object, a different socket is used for each object. Therefore, the OS kernel must search the list of socket descriptors to identify which one is enabled for reading.

4.4 Additional Impediments to CORBA Scalability

In addition to target object demultiplexing overhead, both versions of CORBA used in our experiments possessed other impediments to scalability. In particular, neither worked correctly when clients invoked a large number of operations on a large number of target objects accessed via object references.

We were not able to measure latency for more than ~1,000 objects since both CORBA implementations crashed when we performed a large number of requests on ~1,000 objects. As discussed in Section 4.1, Orbix was unable to support more than ~1,000 objects since it opened a separate TCP connection and allocated a new socket for each object in the server process. Moreover, even though VisiBroker supported ~1,000 objects, it could not support more than 80 requests per object without crashing when the server had 1,000 objects *i.e.,* no more than a total of 80,000 requests could be handled by VisiBroker (this appears to be caused by a memory leak). Clearly, these limitations are not acceptable for mission-critical ORBS.

4.5 Summary of Performance Experiments

The following summarizes our the results of our findings of ORB performance over high-speed networks:

• **Sender-side overhead** Much of the sender-side overhead is in the OS calls used to send requests. Removing this overhead requires the use of optimal buffer manager and tuning different parameters (such as socket queue lengths and flow control strategies) of the underlying transport protocol.

• **Receiver-side overhead** Much of the receiver-side overhead occurs from inefficient demultiplexing and presentation layer conversions (particularly for passing richly-typed data (*e.g.,* `structs`). Eliminating the demultiplexing overhead requires delayered strategies and fast, flexible message demultiplexing [15]. Eliminating the presentation layer overhead requires optimized stub generators [16, 17] for richly-typed data.

• **Demultiplexing overhead** The Orbix demultiplexing performs worse than VisiBroker demultiplexing since Orbix uses a linear search strategy based on `string` comparisons for operation demultiplexing. In addition, due to an open TCP connection for every object reference, Orbix has to use the operating system `select` call to determine the socket descriptors ready for reading.

• **Intra-ORB function calls** Existing ORB implementations suffer from excessive intra-ORB function calls, as shown in Section 4.3. To minimize intra-ORB function calls requires sophisticated compiler optimizations such as integrated layer processing [1].

• **Dynamic invocation overhead** DII performance drops as the size of requests increases. To minimize the dynamic invocation overhead requires allowing reuse of requests and minimizing the marshaling and data copying overhead involved with populating the requests with their intended parameters.

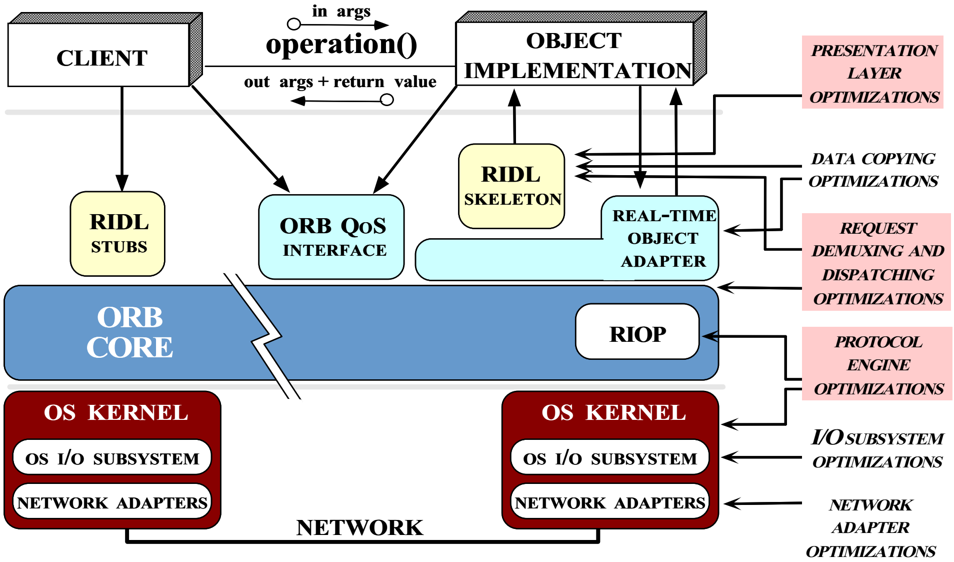

Figure 19: Optimizations for High-Performance, Real-Time ORBs

5 Optimizations for High-Performance ORBs

The performance results reported in this paper reveal the capabilities and limitations of conventional ORBs in terms of latency and scalability. Figure 19 depicts the optimizations we are employing to eliminate the bottlenecks with existing ORBs identified in Section 3. These optimizations are being integrated into a high-performance, real-time ORB called TAO [18, 8].

The research issue we are addressing in TAO is how to provide end-to-end quality of service guarantees to CORBA-compliant applications and services. To accomplish this, we are pursuing an integrated approach that (1) automatically customizes ORBs to take advantage of advanced features offered by operating systems and networks; (2) optimizes generated CORBA IDL stubs to minimize marshaling overhead and to demultiplex incoming requests efficiently; and (3) employs advanced compiler techniques (such as program flow analysis and integrated layer processing) to eliminate excessive intra-ORB function calls.

The central focus of our TAO ORB effort is a portable and feature-rich CORBA kernel that implements the standard CORBA Internet Inter-Operability Protocol (IIOP) as shown in Figure 20. Our IIOP kernel is based on a highly optimized implementation of SunSoft's implementation of the IIOP [19]. It supports the flexible configuration of CORBA-compliant ORBs that can be customized to take advantage of features offered by the underlying network and operating systems (such as resource reservation, zero-copy buffer management, real-time scheduling mechanisms, multi-threading capabilities, and high-speed transport protocols), as well as characteristics of the applications (such as loss tolerance in multimedia applications).

We are developing TAO to overcome the following limita-

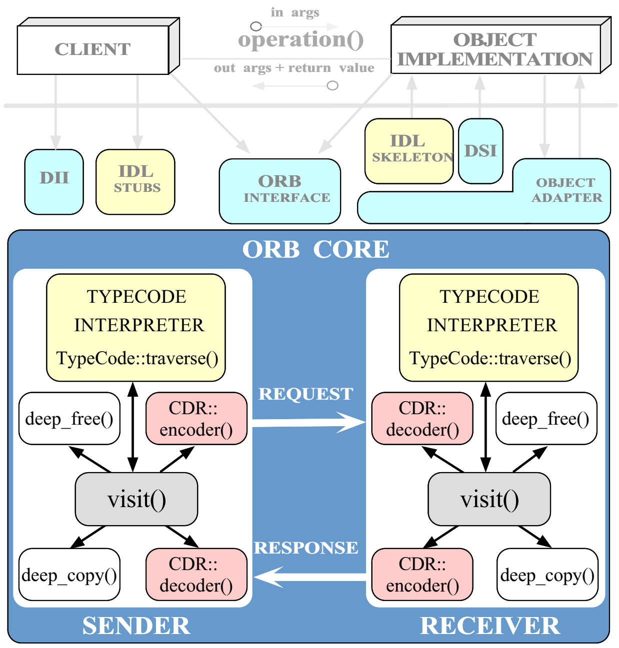

Figure 20: TAO IIOP ORB Core

tions with existing ORBs:

• **Lack of integration with advanced OS and Network features** Existing ORBs (such as Orbix and VisiBroker) do not fully utilize advanced OS and network features. In contrast, the TAO IIOP kernel provides customizable hooks that enable the ORB and CORBA applications to utilize these features, while interoperating seamlessly with IIOP-compliant ORBs.

• **Non-optimal demultiplexing strategies** Existing ORBs utilize inefficient and inflexible demultiplexing strategies based on layered demultiplexing as explained in Section 4.3 and shown in Figure 21(A). In contrast, TAO utilizes active delayered demultiplexing and explicit dynamic linking [14] shown in Figure 21(C), which makes it possible to adapt and configure optimal strategies for dispatching client requests within ORB endsystems and CORBA bridges.

• **Excessive data copying and intra-ORB calls** Existing ORBs are not optimized to reduce the overhead of data copies. In addition, these ORBs suffer from excessive intra-ORB function call overhead as shown in Section 4.3. In contrast, TAO uses advanced compiler techniques (such as program flow analysis [20, 21] and integrated layer processing (ILP)) [1] to automatically omit unnecessary data copies between the CORBA infrastructure and applications. In addition, ILP reduces the overhead of excessive intra-ORB function calls. Most importantly, this streamlining can be performed without requiring modifications to the standard CORBA specification.

• **Inefficient presentation layer conversions** Existing ORBs are not optimized to generate efficient stubs and skeletons. As a result, they incur excessive marshaling and demarshaling overhead ([5, 6] and this paper in Figures 17 and 18). In contrast, TAO produces and configures multiple encoding/decoding strategies for CORBA interface definition language (IDL) descriptions. Each strategy can be configured for different time/space tradeoffs between compiled vs. interpreted CORBA IDL stubs and skeletons [22], and the application's use of parameters (*e.g.*, pass-without-touching, read-only, mutable).

• **Non-optimized buffering algorithms used for network reads and writes** Existing ORBS utilize non-optimized internal buffers for writing to and reading from the network, as shown in Section 4.3. This causes the ORBs to spend a significant amount of time doing reads and writes. In contrast, TAO utilizes optimal buffer choices to reduce this overhead;

We are currently implementing TAO within a prototype real-time OS developed at Washington University. This real-time OS is characterized by features that include (1) a Real-Time Upcall (RTU) scheduling mechanism [23], which achieves end-to-end real-time scheduling and (2) the APIC [24] ATM/host network interface, which provides a zero-copy mechanism that eliminates the excessive data copying overhead. In addition, the ACE framework [25] is used to implement the TAO ORB. ACE contains flexible, reusable, efficient, and portable object-oriented components that automate common ORB communication tasks involving event demultiplexing, event handler dispatching, connection establishment, routing, dynamic configuration of ORB services, and concurrency control.

6 Related Work

Existing research on measuring latency in high performance 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 general, less attention has been paid to integrating the following topics related to communication middleware:

• **Transport Protocol Performance over ATM Networks** The underlying transport protocols used by the ORB must be flexible and possess the necessary hooks to tune different parameters of the underlying transport protocol. [11, 12, 13] present results on performance of TCP/IP (and UDP/IP [11]) 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. [11] shows that UDP performs better than TCP over ATM networks, which is attributed to redundant TCP processing overhead on highly-reliable ATM links. [11] also describes techniques to tune TCP to be a less bulky protocol so that its performance can be comparable to UDP. They also show that the TCP delay characteristics are predictable and that it varies with the throughput.

[26] present detailed measurements of various categories of processing overhead times of TCP/IP and UDP/IP. The authors conclude that whenever a realistic distribution of message sizes is considered, the aggregate costs of non-data touching overheads (such as network buffer manipulation) consume a majority of the software processing time (84% for TCP and 60% for UDP). The authors show that most messages sent are short (less than 200 bytes). They claim that these overheads are hard to eliminate and techniques such as Integrated Layer Processing can be used to reduce the overhead. [27] present performance results of the SUNOS IPC and TCP/IP implementations. They show that increasing the socket buffer sizes improves the IPC performance. They also show that the socket layer overhead is more significant on the receiver side. [28] discusses the `TCP_NODELAY` option, which allows TCP to send small packets as soon as possible to reduce latency.

Earlier work [7, 5, 6] using untyped data and typed data in a similar CORBA/ATM testbed as the one in this paper reveal that the low-level C socket version and the C++ socket wrapper versions of TTCP are roughly equivalent for a given

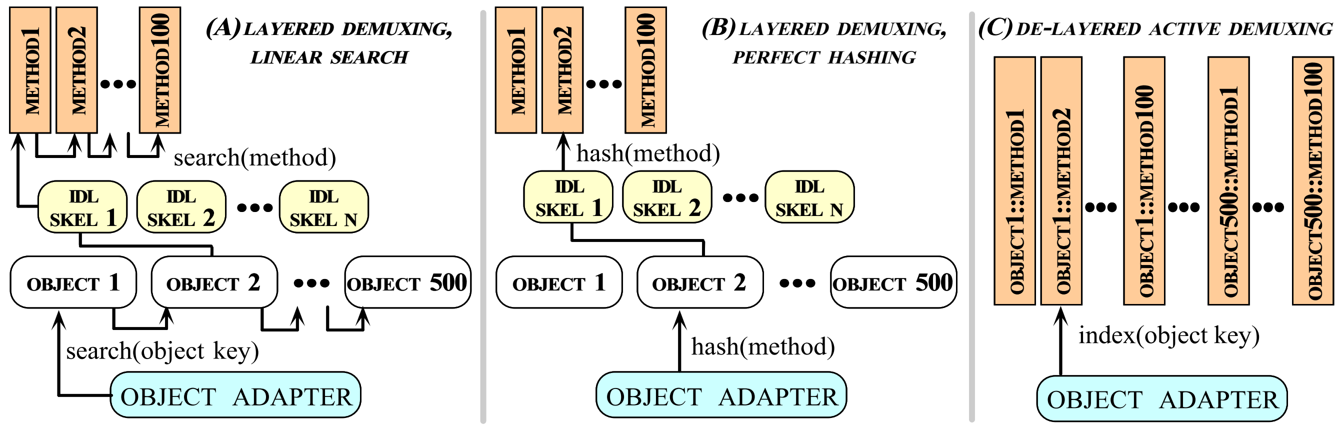

Figure 21: Demultiplexing Strategies

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 socket queue sizes was much worse than those for the scalars. This overhead arises from the amount of time the CORBA implementations spend performing presentation layer conversions and data copying.

The results from this paper reveal that latency increases with increase in number of objects. The variation between request algorithms revealed that the server-side did not cache any information. We plan to incorporate caching behavior in ourTAO ORB to improve latency.

• **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 (marshaling) and vice versa (demarshaling). 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 [29], DCE NDR, or CORBA CDR [2]) to other forms such as a network wire format.

Eliminating the overhead of presentation layer conversions requires highly optimized stub compilers (*e.g.,* Universal Stub Compiler [16]) and the Flick IDL compiler [17]. The generated stub code must make an optimal tradeoff between compiled code (which is efficient, but large in size) and interpreted code (which is slow, but compact) [22].

Our earlier results [5, 6] have presented detailed measurements of presentation layer overhead for transmitting richly-typed data. Our results for sending `structs` reveal that with increasing sender buffer sizes, the marshaling overhead increases, thereby increasing the latency. We plan to design stub compilers that will adapt according to the run-time access characteristics of various data types and operations. The run-time usage of a operation or data type can be used to dynamically link in either the compiled or an interpreted version of marshaling code.

• **Application Level Framing and Integrated Layer Processing on Communication Subsystems** Conventional layered protocol stacks and distributed object middleware lack the flexibility and efficiency required to meet the quality of service requirements of diverse applications running over high-speed networks. One proposed remedy for this problem is to use *Application Level Framing* (ALF) [1, 30, 31] and *Integrated Layer Processing* (ILP) [1, 32, 33]. 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. [34] have shown that although ILP reduces the number of memory accesses, it does not reduce the number of cache misses compared to a carefully designed non-ILP implementation. A major limitation of ILP described in [34] is its applicability to only non-ordering constrained protocol functions and its uses of macros that restrict the protocol implementation from being dynamically adapted to changing requirements.

As shown by our results, CORBA implementations suffer from a number of overheads that includes the many layers of software and large chain of function calls. We plan to use integrated layer processing to minimize the overhead of the various software layers. Our plans call for developing a `factory` of ILP based `inline` functions that are targeted to perform different functions. This way, we can dynamically link in the required function as the requirements change and yet have an ILP based implementation.

• **Demultiplexing** Demultiplexing routes messages between different levels of functionality in layered communication protocol stacks. Most conventional communication models (such as the Internet model or the ISO/OSI reference model) require some form of multiplexing to support interoperability with existing operating systems and protocol stacks. In addition, conventional CORBA implementations utilize several extra levels of demultiplexing at the application layer to associate incoming client requests with the appropriate object implementation and method (as shown in Figure 3). Layered multiplexing and demultiplexing is generally disparaged for high-performance communication systems [35] due to the additional overhead incurred at each layer.[15] describes a fast and flexible message demultiplexing strategy based on dynamic code generation. [14] evaluates the performance of alternative demultiplexing strategies for real-time CORBA.

Our results for latency measurements have shown that with increasing number of objects, the latency increases. This is partly due to the additional overhead of demultiplexing the request to the appropriate method of the appropriate object. We propose to use a delayered demultiplexing architecture [14] that can select optimal demultiplexing strategies based on compile-time and run-time analysis of CORBA IDL interfaces.

7 Concluding Remarks

An important class of applications (such as avionics, distributed interactive simulation, and telecommunication systems) require scalable, low latency communication. As shown in this paper, latency-sensitive applications that utilize many target objects are not yet supported efficiently by contemporary CORBA implementations due to presentation layer conversions, data copying, demultiplexing overhead, and many layers of virtual function calls. 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.

The results presented in this paper illustrate that current implementations of CORBA do not scale well as the number of objects increase by several orders of magnitude. The latency for Orbix for parameterless operations increases roughly 1.12 times for every increase of 100 server objects. In contrast, the latency for VisiBroker remains unaffected due to its hashing-based server-side demultiplexing strategy. However, neither ORB is capable of handling a large number of requests for a large number of objects without crashing.

Latency for sending types data in both ORBS increases as the size of the data increases. This is attributed to the additional overhead of presentation layer conversions. We are currently working towards eliminating presentation layer overhead by using measurement and principle driven optimizations [19]. Our optimizations are based on principles such as eliminating waste, precomputing and storing to avoid unnecessary repetitive computation, and optimizing for the common case.

In general, our latency experiments indicate that CORBA implementations have not been optimized to support low-latency quality of service. Therefore, these CORBA implementations are not yet suited for mission-critical latency-sensitive applications running over high-speed networks. Likewise, our scalability experiments indicate that current CORBA implementations have neither been engineered nor optimized to support large numbers of objects. As a result, these implementations may not yet be suitable to build large-scale, performance-sensitive distributed systems and applications over high-speed networks.

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 for bandwidth-sensitive applications and provide low latency guarantees to delay-sensitive applications. We are currently implementing a lightweight ORB called TAO that eliminates these overheads. The source code for the various tests performed in this paper is made available through the ACE [25] software distribution at `www.cs.wustl.edu/~schmidt/ACE.html`.

Acknowledgments

We like to thank IONA and Visigenic for their help in supplying the CORBA implementations used for these tests. Both companies are currently working to eliminate the latency overhead and scalability limitations described in this paper. We expect their forthcoming releases to perform much better over high-speed ATM networks.

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)*, Philadelphia, PA, Sept. 1990, ACM, pp. 200–208.

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

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

[4] Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, no. 4, November/December 1996.

[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, pp. 306–317.

[6] Aniruddha Gokhale and Douglas C. Schmidt, "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks," in *Proceedings of GLOBECOM '96*, London, England, November 1996, IEEE, pp. 50–56.

[7] 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.

[8] Douglas C. Schmidt, David L. Levine, and Timothy H. Harrison, "An ORB Endsystem Architecture for Hard Real-Time Scheduling," Feb. 1997, Submitted to OMG in response to RFI ORBOS/96-09-02.

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

[10] USNA, *TTCP: a test of TCP and UDP Performance*, Dec 1984.

[11] 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.

[12] 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*, vol. 8, no. 1, pp. 43–48, February 1995.

[13] 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*, Minneapolis, MN, Oct. 1994, IEEE, pp. 176–185.

[14] Aniruddha Gokhale, Douglas C. Schmidt, and Stan Moyer, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *Submitted to GLOBECOM '97*, , Phoenix, AZ, November 1997, IEEE.

[15] Dawson R. Engler and M. Frans Kaashoek, "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation," in *Proceedings of ACM SIGCOMM '96 Conference in Computer Communication Review*, Stanford University, California, USA, August 1996, pp. 53–59, ACM Press.

[16] 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, Aug. 1994.

[17] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, June 1997, ACM.

[18] Douglas C. Schmidt, Aniruddha Gokhale, Tim Harrison, and Guru Parulkar, "A High-Performance Endsystem Architecture for Real-time CORBA," *IEEE Communications Magazine*, vol. 14, no. 2, February 1997.

[19] Aniruddha Gokhale and Douglas C. Schmidt, "Optimizing the Performance of the CORBA Internet Inter-ORB Protocol Over ATM," in *Submitted for publication (Washington University Technical Report #WUCS-97-10)*, February 1997.

[20] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante, "Automatic Construction of Sparse Data Flow Evaluation Graphs," in *Conference Record of the Eighteenth Annual ACE Symposium on Principles of Programming Languages*. ACM, January 1991.

[21] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," in *ACM Transactions on Programming Languages and Systems*. ACM, October 1991.

[22] 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.

[23] R. Gopalakrishnan and G. Parulkar, "A Real-time Upcall Facility for Protocol Processing with QoS Guarantees," in 15^{th} *Symposium on Operating System Principles (poster session)*, Copper Mountain Resort, Boulder, CO, Dec. 1995, ACM.

[24] Zubin D. Dittia, Jr. Jerome R. Cox, and Guru M. Parulkar, "Design of the APIC: A High Performance ATM Host-Network Interface Chip," in *IEEE INFOCOM '95*, Boston, USA, April 1995, IEEE Computer Society Press, pp. 179–187.

[25] Douglas C. Schmidt and Tatsuya Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[26] Jonathan Kay and Joseph Pasquale, "The Importance of Non-Data Touching Processing Overheads in TCP/IP," in *Proceedings of SIGCOMM '93*, San Francisco, CA, September 1993, ACM, pp. 259–269.

[27] Christos Papadopoulos and Gurudatta Parulkar, "Experimental Evaluation of SUNOS IPC and TCP/IP Protocol Implementation," *IEEE/ACM Transactions on Networking*, vol. 1, no. 2, pp. 199–216, April 1993.

[28] S. J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989.

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

[30] 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.

[31] 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.

[32] M. Abbott and L. Peterson, "Increasing Network Throughput by Integrating Protocol Layers," *ACM Transactions on Networking*, vol. 1, no. 5, October 1993.

[33] 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.

[34] Torsten Braun and Christophe Diot, "Protocol Implementation Using Integrated Layer Processnig," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*. ACM, September 1995.

[35] David L. Tennenhouse, "Layered Multiplexing Considered Harmful," in *Proceedings of the* 1^{st} *International Workshop on High-Speed Networks*, May 1989.

A IDL Interface

The following CORBA IDL interface was used by the Orbix and VisiBroker CORBA implementations:

```
struct BinStruct{ short s; char c; long  l;
                  octet o; double d; };
interface ttcp_sequence
{
  typedef sequence<BinStruct> StructSeq;
  typedef sequence<octet> OctetSeq;

  // Routines to send sequences of various data types
  void sendStructSeq_2way (in  StructSeq ttcp_seq);
  void sendOctetSeq_2way (in  OctetSeq ttcp_seq);
  void sendNoParams_2way();
  void sendNoParams_1way();
};
```