Evaluating the Performance of OO Network Programming Toolkits

Timothy H. Harrison and Douglas C. Schmidt
harrison@cs.wustl.edu and schmidt@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130
(314) 935-7538

1 Introduction

For the past several years, the C++ Report has published many articles on the Common Object Request Broker Architecture (CORBA) [1], which is an open standard for distributed object computing (DOC). These articles have focused on the features and components in CORBA that automate common networking tasks such as parameter marshalling, object location, and object activation. This article concentrates on performance issues related to using CORBA and other IPC mechanisms to transfer large amounts of data over low-speed and high-speed networks.

DOC frameworks such as CORBA[1], OODCE[2], and OLE/COM [3] are well-suited for applications that exchange richly typed data via request/response or oneway communication. However, current implementations of DOC frameworks are less suitable for an important class of performance-sensitive applications that stream data over high-speed networks. Medical imaging, interactive teleconferencing, and video-on-demand are common examples of this class of streaming applications.

Streaming applications with stringent throughput and delay requirements are ideal candidates for high-speed networks such as ATM and FDDI. However, these applications may not be able to tolerate the overhead associated with contemporary DOC frameworks. This overhead stems from a number of sources such as non-optimized presentation layer conversions, data copying, and memory management; inefficient and inflexible receiver-side demultiplexing and dispatching operations; synchronous stop-and-wait flow control; and non-adaptive retransmission timer schemes.

Meeting the throughput demands of streaming applications has traditionally required direct access to network programming interfaces such as sockets or System V TLI. These lower-level interfaces are efficient since they omit unnecessary functionality (such as presentation layer conversions for ASCII data). They also allow fine-grained control of memory management, protocol buffering, demultiplexing, and flow control.

However, conventional low-level network programming interfaces are also non-portable and error-prone [4]. This complicates programming and permits subtle run-time errors. For instance, communication endpoints in the socket interface are identified by weakly-typed integer handles (also known as socket descriptors). Weak type-checking increases the potential for run-time errors since compilers cannot detect or prevent improper use of handles. Thus, operations can be applied to handles incorrectly (such as invoking a `read` or `write` on a passive-mode socket handle that can only accept connections).

Developers of high-performance streaming applications have basic programming choices:

1. *Lower-level interfaces* – such as sockets or TLI, which are written in C and are highly efficient.
2. *Middle-level interfaces* – such as the ACE C++ wrappers, Rogue Wave Net.h++, or ObjectSpace System<ToolKit>, which can improve the correctness, ease of use, portability and reusability of communication software without sacrificing much performance.
3. *Higher-level interfaces* – such as DOC frameworks or RPC toolkits, which provide a rich set of abstractions and functionality, but are often highly inefficient for certain types of data.

We've taken representatives from each of the three approaches and conducted experiments on their performance when transfering large streams of data using TCP/IP over Ethernet and ATM networks. The network programming mechanisms compared below include C sockets, ACE C++ wrappers for sockets, and two implementations of CORBA. The benchmark tests are modeled after performance-sensitive applications written by the authors for an enterprise-wide medical imaging system that transports multi-megabyte radiology images across high-speed ATM LANs and WANs [5, 6].

2 Performance Experiments

2.1 Test Platform and Benchmarks

The performance results in this section were collected using a Bay Networks LattisCell 10114 ATM switch connected to a cluster of uni-processor SPARCstation 20 Model 5Os (shown in Figure 1). The LattisCell 10114 is a 16 Port, OC3 155Mbs/port switch. The SPARCstations contain 100 MIP

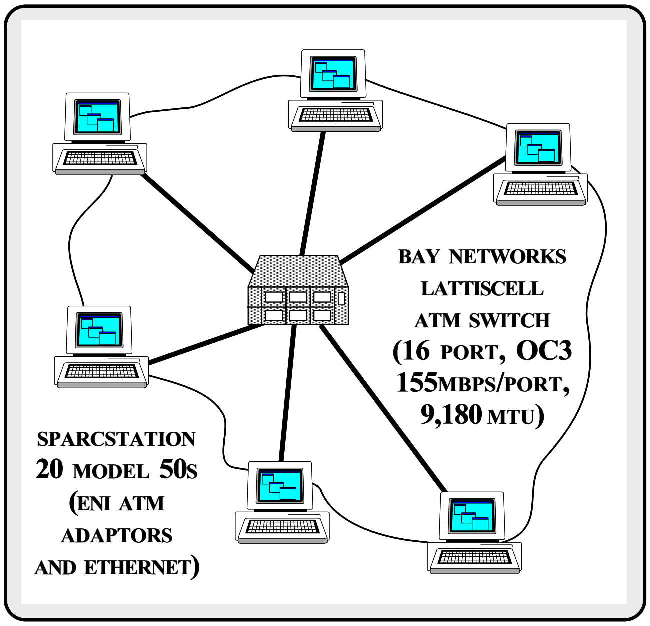

Figure 1: Network/Host Environment for Benchmarks

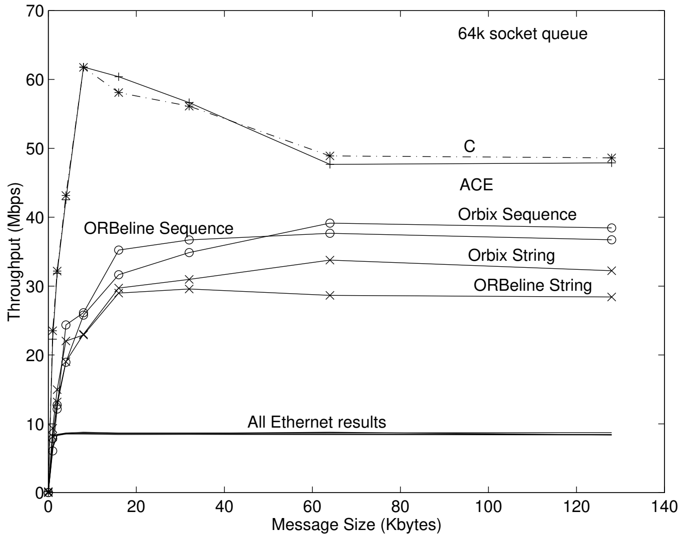

Figure 2: C, ACE, Orbix and ORBeline Performance over ATM and Ethernet

Super SPARC CPUs running SunOS 5.4. The SunOS 5.4 TCP/IP protocol stack is implemented using the STREAMS communication framework [7]. Each SPARCstation 20 has 64 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. 32 Kbytes is alloted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to 8 connections per card.

Data for the experiments was produced and consumed by an extended version of the widely available `ttcp` [8] protocol benchmarking tool. This tool measures end-to-end data transfer throughput in Mbps from a transmitter process to a remote receiver process. The flow of user data is unidirectional, 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 number of data buffers transmitted, the size of data buffers, and the size of the socket transmit and receive queues.

The following versions of `ttcp` were implemented and benchmarked:

- *C version* – this is the standard `ttcp` program implemented in C. It uses C socket calls to transfer and receive data via TCP/IP.
- *ACE C++ version* – this version replaces all C socket calls in `ttcp` with the C++ wrappers for sockets provided by the ACE network programming components (version 3.2) [9]. The ACE wrappers encapsulate sockets with typesafe, portable, and efficient C++ interfaces.
- *CORBA versions* – two implementations of CORBA were used: version 1.3 of Orbix from IONA Technologies and version 1.2 of ORBeline from Post Modern Computing. These versions replace all C socket calls in `ttcp` with stubs and skeletons generated from a pair of CORBA interface definition language (IDL) specifications. One IDL specification uses a `sequence` parameter for the data buffer and the other uses a `string` parameter.

Each version of `ttcp` was compiled using SunC++ 4.0.1 with the highest level of optimization (`-O4`). The timing mechanisms, command-line options, socket options, and communication protocols were held constant for all implementations of `ttcp`. Only the connection establishment and data transfer mechanisms were varied.

2.2 Results

We ran a series of tests that transferred 64 Mbytes of user data in buffers ranging from 1 byte to 128 Kbytes using TCP/IP over Ethernet and ATM networks. Data buffers were run in increments of 1 byte, 1 K, 2 K, 4 K, 8 K, 16 K, 32 K, 64 K, and 128 K sizes. Two different sizes for socket queues were used: 8 K (the default on SunOS 5.4) and 64 K (the maximum size supported by SunOS 5.4).

Figure 2 summarizes the performance results for all the benchmarks using 64 K socket queues over a 155 Mbps ATM link and a 10 Mbps Ethernet (the 8 K socket queue results are presented in Figures 3 and 6). The C and ACE C++ wrapper versions of `ttcp` obtained the highest throughput: 62 Mbps using 8 K data buffers. In contrast, the Orbix and ORBeline CORBA versions of `ttcp` peaked at around 39 Mbps with 64 K data buffers using IDL `sequences`.

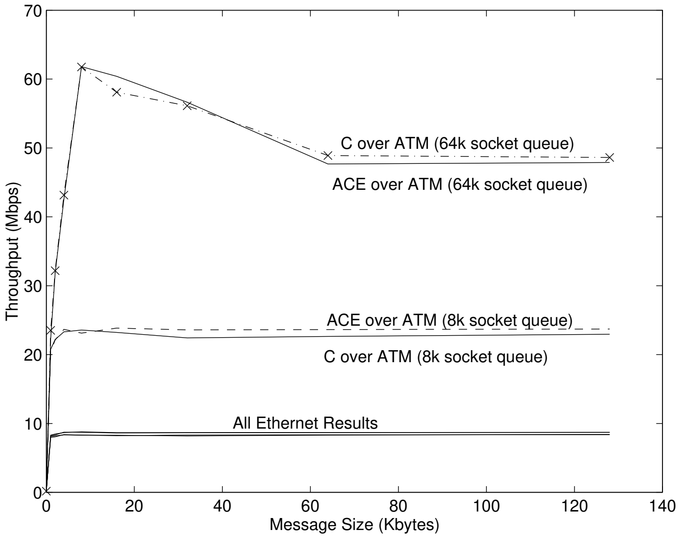

Figure 3: C and ACE Performance over ATM and Ethernet

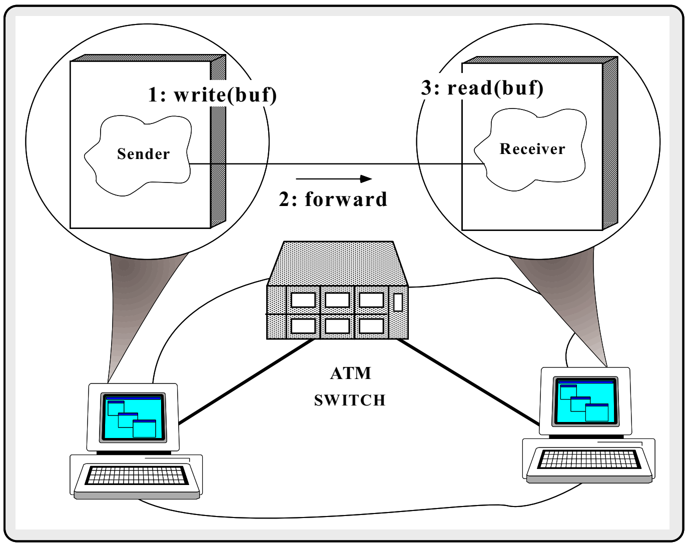

Figure 4: TTCP Configuration for C and ACE C++ Tests

The results for Ethernet show much less variation, with the performance for all tests ranging from around 8 to 8.7 Mbps with 64 K socket queues. None of the Ethernet benchmarks ran faster than 8.7 Mbps, which is 87 percent of the maximum speed of a 10 Mbps Ethernet. Although the absolute throughput of `ttcp` is almost 8 times faster over ATM, the relative utilization of the network channel speed was much lower (*i.e.,* 62 Mbps represents only 40 percent of the 155 Mbps ATM link).

The disparity between network channel speed and end-to-end application throughput is known as the *throughput preservation problem*. This problem occurs when only a portion of the available bandwidth is actually delivered to applications. The throughput preservation problem stems from operating system and protocol processing overhead (such as data movement, context switching, and synchronization). As shown in Section 2.2.2, the throughput preservation problem is exacerbated by contemporary implementations of DOC frameworks like CORBA, which copy data multiple times during fragmentation/reassembly, marshalling, and demarshalling.

Sections 2.2.1 and 2.2.2 examine these performance results in detail and Section 3 presents recommendations based on an analysis of the benchmark results.

2.2.1 C and ACE C++ Wrapper Implementations of TTCP

Figure 4 shows the configuration of the `ttcp` driver for the C and ACE C++ wrapper tests. For both these tests, the sender process writes 64 Mbps of data directly to the socket layer, which forwards it across the network to the receiver process. The receiver process reads the data from the socket and "consumes" it.[1]

Figure 3 illustrates the performance results from the C and ACE wrapper versions of `ttcp` over ATM and Ethernet. The performance of C sockets and ACE C++ wrappers are roughly equivalent. Both peak at 62 Mbps over ATM using 8 K data buffers and 64 K socket queues. This indicates that the performance penalty for using the ACE C++ wrappers is insignificant, compared with using C library function calls directly.

Figure 3 illustrates the impact of data buffer size on performance. When the data buffers exceeded 8 K performance began to decline, leveling off at around 48 Mbps with 64 K data buffers. This behavior is caused primarily by the MTU size of the ATM network, which is 9,180 bytes. When data buffers exceed the MTU size they are fragmented and reassembled, thereby lowering performance.

Figure 3 also illustrates the impact of socket queue size on throughput. Larger socket queues increase the TCP window size, which allows the transmission of multiple TCP segments back-to-back. In the case of ATM, increasing the socket queue from 8 K to 64 K improves `ttcp` performance significantly from 23 Mbps to 62 Mbps.

The Ethernet results for large and small socket queues show less variation than the ATM results. They peak at 8.4 Mbps with 8 K socket queues and 8.7 Mbps with 64 K socket queues. In both cases, the factor limiting performance is the slow speed of the network.

2.2.2 CORBA Implementations of TTCP

Figure 5 shows the configuration of `ttcp` for the CORBA tests. For both these tests, the sender writes 64 Mbps to the receiver by invoking the `send` method on the `TTCP` stub. The stub forwards the data across the network to the ORB on

[1] Since these tests are only measuring network performance, the receiver just records the number of bytes received and doesn't actually process the data.

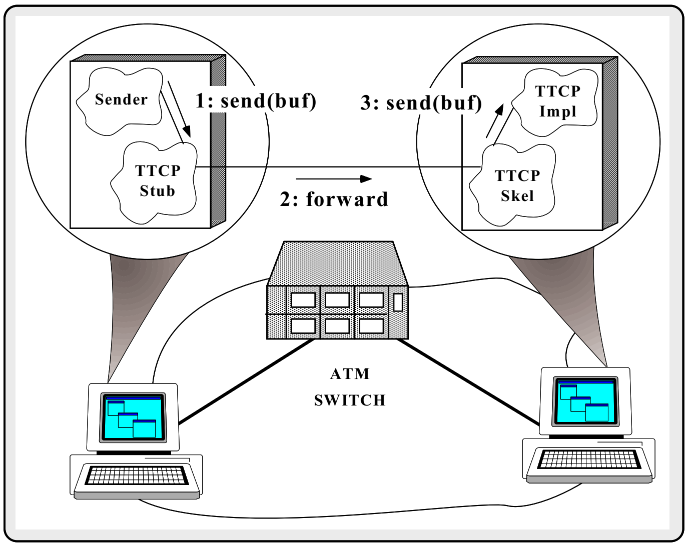

Figure 5: TTCP Configuration for the CORBA Tests

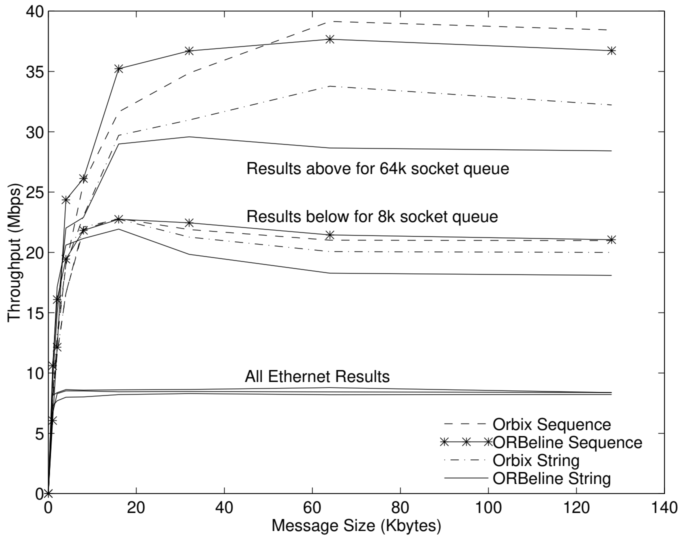

Figure 6: Orbix and ORBeline Performance over ATM and Ethernet

the server. The ORB's Object Adapter then performs demultiplexing operations to locate the `TTCP` skeleton. Finally, this skeleton passes the data up to the `TTCP` implementation object by invoking a `send` upcall. As before, the receiver just records the number of bytes received, rather than processing the data.

Figure 6 illustrates the results of measuring two versions of `ttcp` implemented with two different versions of CORBA. The CORBA implementations were developed using single-threaded versions of Orbix 1.3 and ORBeline 1.2.

Extending `ttcp` to use CORBA required several modifications to the original C/socket code. All C socket calls were replaced with stubs and skeletons generated from a pair of CORBA interface definitions. One IDL interface uses a `sequence` to transmit the data and the other IDL interface uses a `string`, as follows:

```
typedef sequence<char> ttcp_sequence;

interface TTCP_Sequence
{
  oneway void send (in ttcp_sequence ttcp_seq);
};

interface TTCP_String
{
  oneway void send (in string ttcp_string);
};
```

The `send` operations use `oneway` semantics since the `ttcp` benchmarks measure the performance of uni-directional data transfer. This behavior is consistent with the flow of communication in electronic medical imaging applications and video distribution.

The client-side of `ttcp` was modified as follows:

```
// Use locator service to acquire bindings.
TTCP_String *t_str = TTCP_String::_bind ();
TTCP_Sequence *t_seq = TTCP_Sequence::_bind ();
```

The `_bind` method is a factory generated by the IDL compiler from an IDL specification (such as `TTCP_Sequence` and `TTCP_String`). This factory obtains CORBA *object references* to object implementations of `TTCP_Sequence` and `TTCP_String` located on a server. Object references are opaque, immutable handles that uniquely identify objects. All CORBA object implementations must have an object reference before they can be accessed by client applications, and all client applications must have an object reference before they can access the object implementations in the server.

Once object references are obtained, data buffers of the appropriate size can be initialized and transmitted by calling the IDL-generated `send` stubs, as follows:

```
// String transfer.

char *buffer = new char[buffer_size];
// Initialize data in char * buffer...

while (--buffers_sent >= 0)
  t_str->send (buffer);

// Sequence transfer.

ttcp_sequence sequence_buffer;
// Initialize data in TTCP_Sequence buffer...

while (--buffers_sent >= 0)
  t_seq->send (sequence_buffer);
```

The server-side was modified to create object implementations for `TTCP_Sequence` and `TTCP_String`. CORBA IDL compilers generate skeletons that translate IDL interface definitions (such as `TTCP_Sequence`) into C++ base classes (such as `TTCP_SequenceBOAImpl`). Each IDL operation (such as `oneway void send`) is mapped to a corresponding C++ pure virtual method (such as `virtual void send`). Programmers then define C++ derived classes that override these virtual methods to implement application-specific functionality, as follows:[2]

```
// Implementation class for IDL interface
// that inherits from automatically-generated
// CORBA skeleton class.

class TTCP_Sequence_i
  : virtual public TTCP_SequenceBOAImpl
{
public:
  TTCP_Sequence_i (void): nbytes_ (0) {}

  // Upcall invoked by the CORBA skeleton.
  virtual void send
    (const ttcp_sequence &ttcp_seq,
     CORBA::Environment &IT_env)
  {
    this->nbytes_ += ttcp_seq._length;
  }
  // ...

private:
  // Keep track of bytes received.
  u_long nbytes_;
};
```

The server-side uses the CORBA `impl_is_ready` event loop to demultiplex incoming requests to the appropriate object implementation, as follows:

```
int main (int argc, char *argv[])
{
  // Implements the Sequence object.
  TTCP_Sequence_i ttcp_sequence;

  // Implements the String object.
  TTCP_String_i ttcp_string;

  // Single-threaded event loop that handles
  // CORBA requests by making callbacks to
  // user-supplied object implementations
  // of TTCP_Sequence_i and TTCP_String_i.
  CORBA::BOA::impl_is_ready ();

  /* NOTREACHED */
  return 0;
}
```

By comparing Figure 6 with Figure 3 it is clear that the CORBA-based `ttcp` implementations ran considerably slower than the C and ACE wrapper versions on the ATM network, particularly for 8 K data buffers. The highest throughput (39 Mbps) was obtained by the Orbix `sequence` implementation using 64 K data buffers and 64 K socket queues. The throughput leveled off beyond 64 K data buffers.

Unlike the C and ACE wrapper results in Figure 2, the performance of the CORBA versions did not decrease when the size of the data buffers exceeded 8 K. This behavior stems from the higher fixed overhead of CORBA (such as demultiplexing and memory management) that lowers its performance for small buffer sizes. As the buffer size increases, the relative impact of this fixed overhead is reduced. However, as the size of the buffers increase so does the overhead of data copying. As shown below, data copying ultimately limits the throughput achievable with the CORBA implementations.

Test	%Time	#Calls	msec/call	Name
C sockets (sender)	99.6	527	92.8	_write
C sockets (receiver)	99.3	7201	6.2	_read
ACE C++ wrapper (sender)	99.4	527	87.3	_write
ACE C++ wrapper (receiver)	99.6	7192	6.2	_read
Orbix Sequence (sender)	94.6	532	89.1	_write
	4.1	2121	1.0	memcpy
Orbix Sequence (receiver)	92.7	7860	6.1	_read
	4.8	2581	0.6	memcpy
Orbix String (sender)	89.0	532	85.6	_write
	4.6	2121	1.1	memcpy
	4.1	2700	0.7	strlen
Orbix String (receiver)	86.3	7744	5.7	_read
	5.5	6740	0.4	strlen
	4.5	2581	0.9	memcpy
ORBeline Sequence (sender)	91.0	551	74.9	_write
	5.2	6413	0.4	memcpy
	1.8	1032	0.8	__sigaction
ORBeline Sequence (receiver)	89.0	7568	5.8	_read
	5.1	7222	0.3	memcpy
	3.3	1071	1.5	_poll
ORBeline String (sender)	83.8	551	83.9	_write
	5.4	920	3.2	strcpy
	4.3	5901	0.4	memcpy
	3.9	1728	1.2	strlen
	1.1	1032	0.6	__sigaction
ORBeline String (receiver)	85.4	7827	5.5	_read
	4.6	6710	0.3	memcpy
	4.2	1702	1.3	strlen
	2.8	1071	1.3	_poll

Figure 7: High cost Functions for `ttcp` Tests

Detailed profiling and examination of the IDL stubs and skeletons generated by Orbix and ORBeline revealed that the CORBA overhead stems from the following sources:

• **Data Copying:** The data buffers exchanged between the sender and receiver in `ttcp` are treated as a stream of untyped bytes. This is consistent with the type of data transmitted by high-performance streaming applications like teleconferencing and medical imaging. In a sense, the tests reported here represent the "best" case for the CORBA implementations. Since the data is untyped, the CORBA presentation layer doesn't need to perform complex marshalling to handle byte-ordering differences between sender and receiver.[3]

Although marshalling was not required for our tests, the CORBA implementations still incurred significant data copying overhead. We used the UNIX execution profiler (`prof`) to pinpoint the sources of this overhead. The C++ compiler was directed to instrument the source code with monitoring instructions and `prof` was then used to measure the amount of time spent in functions during program execution. Figure 7

[2] Both CORBA implementations of `ttcp` used inheritance because ORBeline 1.2 din't support Orbix's "TIE" technique, which uses object composition to associate application-specific CORBA class implementations to the generated IDL skeletons. Both Orbix and ORBeline 2.0 now support the "TIE" technique.

[3] When when transfering richly-typed data instead of the untyped-data used in the `ttcp` tests, CORBA implementations incur much higher overhead, achieving only around 30 percent of the performance of hand-crafted C and C++ marshalling [10].

lists the functions where the most time was spent sending and receiving 64 Mbytes using 128 K data buffers and 64 K socket queues.

The `read` and `write` system calls accounted for more than 99% of the execution time in the C and ACE C++ wrapper implementations of `ttcp`. Note that although the data was transmitted as 512 separate 128 K buffers it was read by the receiver in much smaller chunks of around 8 K. This illustrates the fragmentation and reassembly performed by the ATM network adaptors (whose MTU is 9,180 bytes).

The `read` and `write` system calls dominated the execution of the CORBA implementations, as well. Unlike the C and ACE wrapper versions, however, these implementations spent 4 to 15 percent of their time performing other tasks, such as copying and/or inspecting data (`memcpy`, `strcpy`, and `strlen`), checking for activity on other I/O handles (`_poll`), and manipulating signal handlers (`__sigaction`). This shows that the highest cost tasks involved data copying and data inspection. The IDL stubs and skeletons copy data multiple times (*e.g.,* from the TCP data buffer into a marshalling buffer, and then again into the parameter passed to the `send` upcall).

The test results also illustrate that the choice of CORBA IDL parameter datatypes has a significant impact on performance. The `sequence` implementations shown in Figure 6 peaked at 39 Mbps for Orbix and 38 Mbps for ORBeline. In contrast, the `string` implementations peaked at 34 Mbps for Orbix and 30 Mbps for ORBeline. The performance variation between the `sequence` and `string` results are due to differences in their IDL-to-C++ mappings. In particular, the IDL `sequence` mapping contains a length field, whereas the `string` mapping does not. The generated IDL stubs and skeletons use this length field to avoid searching each `sequence` parameter for a terminating NUL character. In contrast, the IDL `string` implementations use `strlen` to determine the length of their parameters.

The performance variation between Orbix and ORBeline is partially explained by the differences in their message fragmentation/reassembly implementations, as well as the design of their socket event handling. As shown in Figure 7, ORBeline copies data approximately 3 more times than Orbix on the sender and receiver for both `sequence` and `string`.

In addition, ORBeline invokes the `sigaction` and `poll` system calls twice for each message that is sent and received, respectively. The `sigaction` call disables the `SIGPIPE` signal during a `write` system call. On most UNIX systems the default behavior on `SIGPIPE` is to exit the program. `SIGPIPE` occurs when data is sent over a socket whose peer has reset the connection. To unobtrusively prevent this from happening, ORBeline 1.2 replaces any existing handlers with `SIG_IGN` disposition before the `write` and resets it to the original disposition following the `write`. The Orbix implementation does not perform these operations, which is one reason why ORBeline's throughput was consistently lower than Orbix (as shown in Figure 6).[4]

[4] ORBeline 2.0 eliminates this additional signal handler overhead.

• **Demultiplexing:** Each CORBA request message contains the name of its intended remote operation, which is represented as a string. Orbix demultiplexes incoming messages to the appropriate upcall by performing a linear search through the list of operations in the IDL interface. In the case of `ttcp`, linear search suffices since there was only one choice (`send`). However, this strategy doesn't scale well since search time grows linearly with the number of operations in the IDL interface. Moreover, the order of operations will determine the demultiplexing performance. Therefore, operations in Orbix should be ordered by decreasing frequency of use.

In contrast, ORBeline use hashing to determine the appropriate upcall associated with an incoming request. Hashing is likely to scale better for large IDL interfaces, but may be less efficient for small interfaces due to the overhead of computing the hash function. To handle these and other cases efficiently, the demultiplexing of requests can benefit from *adaptive* optimizations. These optimizations select customized strategies depending on the properties of the IDL interface. For example, perfect hashing or some type of integral indexing scheme could be negotiated between sender and receiver to improve performance and to shield developers from having to manually tune their IDL interfaces.

• **Memory allocation:** IDL skeletons generated automatically by a CORBA IDL compiler do not know how the user-supplied upcall will use the parameters passed to it from the request message. Thus, they use conservative memory management techniques that dynamically allocate and release copies of messages before and after an upcall, respectively. These memory management policies are important in some circumstances (*e.g.,* if an upcall is used in a multi-threaded application). However, this strategy needlessly increases processing overhead for streaming applications like `ttcp` that consume their data immediately without modifying it.

3 Evaluation and Recommendations

Section 2.2 compared the performance of C, ACE wrapper, and CORBA versions of `ttcp` in terms of their ability to transfer large quantities of data using TCP/IP over Ethernet and ATM networks. In this section, we evaluate these results and present recommendations for using DOC frameworks over high-speed networks.

• **Know your requirements:** It's important to evaluate tools based on empirical measurements and a realistic understanding of application requirements, rather than adopting a particular communication model or implementation unconditionally. Although we focus on performance, your choice of communication mechanisms should consider other factors, as well. For instance, learning curve, cost of tools and run-time licenses, the ability to rapidly prototype, and time to market are just some of the requirements that guide the selection of a particular communication mechanism.

If performance is your only requirement, using low-level mechanisms may be the best choice. At the moment, the C-level socket API or C++ wrappers for sockets will provide higher throughput than DOC frameworks over TCP/IP. If that's still not enough, and you don't need reliability or portability, bypass TCP/IP and program directly to the high-speed data link layer (e.g., ATM or FDDI) and/or get hardware support for your data streams. This approach is often used in high-speed video-on-demand systems [11].

On the other hand, if your requirements call for flexibility, maintainability, and reusability, you should consider a higher-level DOC framework such as CORBA or Network OLE. Although the performance will be lower, these DOC frameworks are valuable since they automate many common network programming tasks such as object selection, location, and activation, as well as parameter marshalling and framing.

• **Know your environments:** If you have a network in place, use a `ttcp` test to measure its maximum achievable throughput.[5] If your project has already deployed a software communication framework, test its throughput. Compare what you have with what is possible. You will undoubtedly find that the overhead added by the communication framework (*i.e.,* CORBA, RPC, etc.) is just one aspect of high-performance communication. In particular, the overhead of the OS and network adapters is often much more significant than the overhead of CORBA in well-tuned ORBs. Ultimately, the best solution is an integrated approach that attacks performance overhead at multiple levels of abstraction (e.g., network adapter, OS kernel, transport protocols, demultiplexing, presentation layer, data copying, etc.).

If your applications don't push the limits of your network's potential (and it's likely they won't unless you're on a low-speed Ethernet or Token-Ring) consider the next step. If your requirements call for better performance, and you don't have the time, resources, or inclination to develop your own communication infrastructure, CORBA is a good choice. Good CORBA implementations will achieve reasonable performance with minimal development time.

• **Break the golden rule of abstraction:** One of the often cited benefits of CORBA is that it abstracts away from the lower-layer networking details. Therefore, it's tempting to think that we can write applications using CORBA and never worry about what's going on down below. Unfortunately, things aren't quite this easy when performance-sensitive applications are run over high-speed networks. In this case, you'll inevitably need access to lower-level mechanisms to maximize your application performance.

For example, it's particularly important to increase the size of the socket queues to the largest values supported by the OS. This is illustrated by the considerable difference in throughput for the 8 K and 64 K socket queues in Figures 3 and 6. In fact, this figure shows that the choice of socket queue size has more impact than the choice of communication model (*i.e.,* C/C++ vs. CORBA). The slowest communication model (CORBA) is faster with 64 K socket queues than the faster communication model (C/C++) with 8 K queues.

• **Demand that CORBA implementors provide hooks:** DOC frameworks that allow access to low-level mechanisms (such as socket queues) are destined to perform poorly when used over high-speed networks. For this reason, Orbix provides hooks to enlarge socket queues via `setsockopt` by invoking a user-defined callback function whenever a new socket is connected. Likewise, ORBeline 2.0's event handling mechanism gives clients and servers access to the ORB at a lower level (including access to connection handles). Unfortunately, not all ORBs provide these hooks yet, so make sure to check with your ORB vendors before committing to a particular product.

The need for hooks also surfaces when trying to integrate multiple event loops. ORB event loops must be accessible if developers are to use CORBA for large-scale projects that involve other tools (such as GUI event loops, which typically use their own event loops [6]). Thus, ORB event loops must either be able to surrender control to another event loop or assume control of event loops running in the same process space. The most stubborn event loop typically wins. For instance, on Windows platforms the GUI event loop is tightly coupled with the operating system, which makes it hard for an ORB event loop to assume full control.

Orbix allows users to replace the default event loop through the use of `CORBA::Orbix.registerIOCallback` and `CORBA::Orbix.processNextEvent`. `CORBA::Orbix.registerIOCallback` gives users access to the socket descriptors being used by the ORB. `CORBA::Orbix.processNextEvent` allows the ORB to process a single incoming request. By combining the two, developers can easily integrate Orbix with their own `select` loop. Orbix integrates with the Windows event loop internally. However, developers must explicitly integrate with the X Windows event loop using `registerIOCallback` and `processNextEvent`. Orbix is shipped with an Xwin demo that shows how this can be done.

For those who want to integrate ORBeline with Graphical User Interfaces, almost no development needs to be done. ORBeline offers different implementations of their Dispatcher event loop. These include an XDispatcher for X Windows, a WinDispatcher for Windows and Windows NT, and even a GalaxyDispatcher. For more intimate control, developers can use `Dispatcher::dispatch` in combination with ORBeline event handlers. `Dispatcher::dispatch` tells the ORB to process the next incoming request, and ORBeline event handlers can be used to access the socket descriptors used by the ORB. For even more control, ORBeline also allows you to replace wholesale the implementation of the

[5] The `ttcp` benchmarks and ACE source code described in this paper are freely available and may be obtained via the WWW at URL `http://www.cs.wustl.edu/ schmidt/ACE.html`. We encourage others to replicate our `ttcp` experiments using different implementations of CORBA and other network/host platforms and report the results.

Dispatcher used by their ORB.

• **Understand CORBA implementation issues:** As users and organizations migrate to high-speed networks, the performance limitations of contemporary CORBA implementations will become more evident. Hopefully, this will encourage vendors to optimize their ORBs for streaming performance-sensitive applications running over high-speed networks. Key areas of optimization include data copying and data inspection, presentation layer conversions, memory management, and receiver-side demultiplexing and dispatching [10]. In particular, implementations must reduce the number of times that large data buffers are copied in the endsystems.

However, users can take steps to increase the performance of their ORBs. In particular, you can use IDL constructs that minimize client and server stub marshalling requirements (*e.g.* use `sequences` rather than `strings`). Although this violates some of the fundamental abstractions of CORBA, it's still easier than programming at the socket level. Idealism aside, if it works, use it.

• **Mix and match:** Until CORBA optimizations are widely implemented in production systems, we need to find solutions for today. Our solution has been to integrate higher-level DOC frameworks with high-performance object-oriented encapsulations of lower-level network programming interfaces (such as the ACE C++ wrappers for sockets described in [4]). We've built a framework that combines CORBA and the ACE C++ wrappers and used it for a production high-speed teleradiology system that transfers large, multi-Mbyte medical images over ATM [6]. In this system, CORBA is used as a signaling mechanism to identify endpoints of communication in a location-independent manner. The ACE C++ wrappers are then used to establish point-to-point TCP connections and transmit bulk data efficiently across the connections. This strategy builds on the strengths of both CORBA and ACE.

4 Concluding Remarks

An important class of applications require high-performance streaming communication. Bandwidth-intensive and delay-sensitive streaming applications like medical imaging or teleconferencing are not supported efficiently by contemporary CORBA implementations due to data copying, demultiplexing, and memory management overhead. As shown in Section 2, this overhead is often masked on low-speed networks like Ethernet and Token Ring. On high-speed networks like ATM or FDDI, however, this overhead becomes a significant factor limiting communication performance.

The performance results and recommendations in this paper are not intended as a criticism of the CORBA model or of particular ORB vendors. It is beyond the scope of this paper to discuss the benefits (such as extensibility and maintainability) of CORBA, as well as its limitations. We would 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 networks.

References

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

[2] J. Dilley, "OODCE: A C++ Framework for the OSF Distributed Computing Environment," in *Proceedings of the Winter Usenix Conference*, USENIX Association, January 1995.

[3] Microsoft Press, Redmond, WA, *Object Linking and Embedding Version 2 (OLE2) Programmer's Reference, Volumes 1 and 2*, 1993.

[4] D. C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

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

[6] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "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), USENIX, June 1996.

[7] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

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

[9] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[10] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), ACM, August 1996.

[11] J. R. Cox, W. D. Richard, K. Krieger, and B. Gottlieb, "The Washington University Multimedia System," in *ACM Multimedia System*, pp. 120–131, Springer-Verlag, Jan. 1993.