Object-Oriented Components for High-speed Network Programming

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

An earlier version of this paper appeared in the proceedings of the USENIX Conference on Object-Oriented Technologies, Monterey, CA, June 1995.

Abstract

This paper makes two contributions to the development and evaluation of object-oriented communication software. First, it reports performance results from benchmarking several network programming mechanisms (such as sockets and CORBA) on Ethernet and ATM networks. These results illustrate that developers of high-bandwidth, low-delay applications (such as interactive medical imaging or teleconferencing) must evaluate their performance requirements and the efficiency of their communication infrastructure carefully before adopting a distributed object solution. Second, the paper describes the software architecture and design principles of the ACE object-oriented network programming components. These components encapsulate UNIX and Windows NT network programming interfaces (such as sockets, TLI, and named pipes) with C++ wrappers. Developers of object-oriented communication software have traditionally had to choose between high-performance, lower-level interfaces provided by sockets or TLI or less efficient, higher-level interfaces provided by communication frameworks like CORBA or DCE. ACE represents a midpoint in the solution space by improving the correctness, programming simplicity, portability, and reusability of performance-sensitive communication software.

1 Introduction

Distributed object computing (DOC) frameworks like the Common Object Request Broker Architecture (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 may be less suitable for an important class of performance-sensitive applications that stream relatively simple datatypes 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 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 [4] or System V TLI [5]. 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 network programming interfaces are low-level, non-portable, and non-typesafe. 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).

Traditionally, developers of high-performance streaming applications had to choose between two solutions:

1. *Higher-level, but less efficient network programming interfaces* – such as DOC frameworks or RPC toolkits;
2. *Lower-level, but more efficient network programming interfaces* – such as sockets or TLI.

This paper describes object-oriented network programming components that provide a midpoint in the solution space. These components are part of the ACE toolkit [6], which encapsulates conventional network programming interfaces with a family of C++ wrappers. As shown below, the ACE toolkit improves the correctness, ease of use, portability and reusability of communication software without sacrificing performance.

This paper is organized as follows: Section 2 compares the performance of several network programming mechanisms (C sockets, C++ wrappers for sockets, and two implementations of CORBA) for a representative streaming application over Ethernet and ATM networks; Section 3 outlines the design of the object-oriented ACE components that encapsulate UNIX and Windows NT network programming interfaces (such as sockets, TLI, STREAM pipes, and named pipes); Section 4 illustrates the differences between programming with C sockets, ACE, and CORBA; Section 5 summarizes the design principles of the ACE wrappers; and Section 6 presents concluding remarks.

2 Performance Experiments

This section describes performance results from comparing several network programming mechanisms that transfer large streams of data using TCP/IP over Ethernet and ATM networks. The network programming mechanisms compared below include C sockets, C++ wrappers for sockets, and two implementations of CORBA. The benchmark tests are representative of applications written by the authors for the Motorola Iridium project [7] and Project Spectrum [8]. Iridium is a next-generation satellite-based global personal communication system; Spectrum is an enterprise-wide medical imaging system that transports radiology images across high-speed ATM LANs and WANs.

2.1 Test Platform and Benchmarks

The performance results in this section were collected using a Bay Networks LattisCell 10114 ATM switch connected to two uni-processor SPARCstation 20 Model 5Os. The LattisCell 10114 is a 16 Port, OC3 155Mbs/port switch. The SPARCstations contain 100 MIP Super SPARC CPUs running SunOS 5.4. The SunOS 5.4 TCP/IP protocol stack is implemented using the STREAMS communication framework [9]. 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 64K). 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` [10] 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 uni-directional, with the transmitter flooding the receiver with a user-specified number of data buffers. Various sender and receiver parameters may be selected at run-time. These parameters include the 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 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) [6]. 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`). To control for confounding factors, 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, 1K, 2K, 4K, 8K, 16K, 32K, 64K, and 128K sizes. Two different sizes for socket queues were used: 8K (the default on SunOS 5.4) and 64K (the maximum size supported by SunOS 5.4). Each test was run 20 times to account for performance variation due to transient load on the networks and hosts. The variance between runs was very low since the tests were conducted on otherwise unused networks.

Figure 1 summarizes the performance results for all the benchmarks using 64K socket queues over a 155 Mbps ATM link and a 10 Mbps Ethernet (the 8K socket queue results are presented in Figures 2 and 3 and Tables 1 and 2 summarize the results for all the tests). The C and ACE C++ wrapper versions of `ttcp` obtained the highest throughput: 62 Mbps using 8K data buffers. In contrast, the Orbix and ORBeline CORBA versions of `ttcp` peaked at around 39 Mbps with 64K data buffers using IDL `sequences`.

The results for Ethernet show much less variation, with the performance for all tests ranging from around 8 to 8.7 Mbps with 64K 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

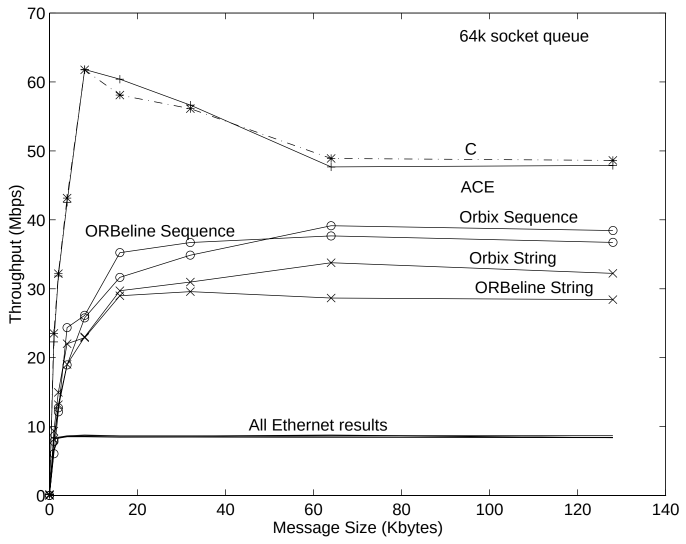

Figure 1: C, ACE, Orbix and ORBeline Performance over ATM and Ethernet

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* [11]. 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 [12]). 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 2.3 presents recommendations based on an analysis of the benchmark results.

2.2.1 C and ACE Wrapper Implementations of TTCP

Figure 2 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 8K data buffers and 64K 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 2 illustrates the impact of data buffer size on performance. When the data buffers exceeded 8K performance began to decline, leveling off at around 48 Mbps with 64K 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.

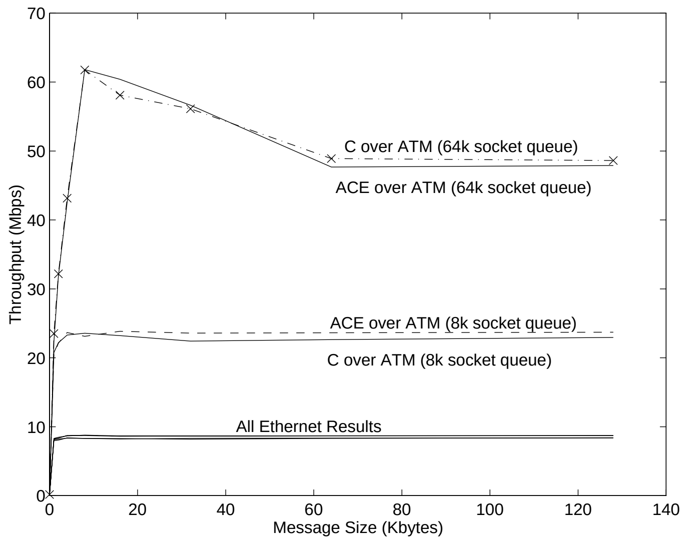

Figure 2: C and ACE Performance over ATM and Ethernet

Figure 2 also illustrates the impact of socket queue size on throughput. Larger socket queues increase the TCP window size [13], which allows the transmission of multiple TCP segments back-to-back. In the case of ATM, increasing the socket queue from 8K to 64K 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 8K socket queues and 8.7 Mbps with 64K socket queues. In both cases, the factor limiting performance is the slow speed of the network.

2.2.2 CORBA Implementations of TTCP

Figure 3 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. At the time these tests were performed, neither Orbix nor ORBeline fully supported the OMG 2.0 CORBA standard. This complicated the CORBA implementations of `ttcp` since different versions were required to account for incompatibilities between Orbix and ORBeline.

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);
};
```

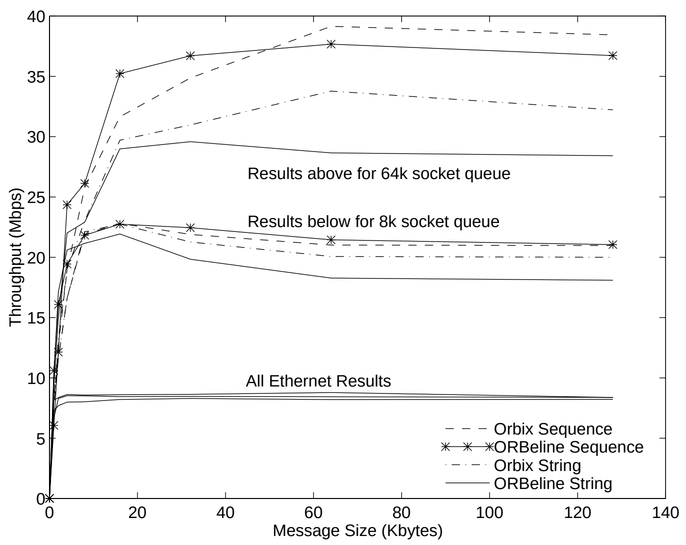

Figure 3: Orbix and ORBeline Performance over ATM and Ethernet

```
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 *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 one 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 the object references were obtained, data buffers of the appropriate size were initialized and then 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:[1]

```
// 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 used 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;
}
```

Porting `ttcp` to use CORBA over ATM demonstrated the importance of having hooks to manipulate underlying OS mechanisms (such as transport layer and socket layer

[1]Both CORBA implementations of `ttcp` used inheritance since ORBeline does not support Orbix's "TIE" technique (which uses object composition to associate application-specific CORBA class implementations to the generated IDL skeletons).

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	94.6	532	89.1	_write
(sender)	4.1	2121	1.0	memcpy
Orbix Sequence	92.7	7860	6.1	_read
(receiver)	4.8	2581	0.6	memcpy
Orbix String	89.0	532	85.6	_write
(sender)	4.6	2121	1.1	memcpy
	4.1	2700	0.7	strlen
Orbix String	86.3	7744	5.7	_read
(receiver)	5.5	6740	0.4	strlen
	4.5	2581	0.9	memcpy
ORBeline Sequence	91.0	551	74.9	_write
(sender)	5.2	6413	0.4	memcpy
	1.8	1032	0.8	__sigaction
ORBeline Sequence	89.0	7568	5.8	_read
(receiver)	5.1	7222	0.3	memcpy
	3.3	1071	1.5	_poll
ORBeline String	83.8	551	83.9	_write
(sender)	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	85.4	7827	5.5	_read
(receiver)	4.6	6710	0.3	memcpy
	4.2	1702	1.3	strlen
	2.8	1071	1.3	_poll

Figure 4: High cost Functions for `ttcp` Tests

options) that significantly affect performance. In particular, high performance data transfers over TCP and ATM require large socket queues. This is illustrated by the considerable difference in throughput for the 8K and 64K socket queues in Figures 2 and 3.

Orbix provides hooks to enlarge socket queues via `setsockopt` by invoking a user-defined callback function whenever a new socket is connected. In contrast, it was hard to enlarge the socket queues using ORBeline 1.2 since it did not provide direct access to sockets (subsequent versions of ORBeline will provide this functionality).

By comparing Figure 3 with Figure 2 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 8K data buffers. The highest throughput (39 Mbps) was obtained by the Orbix `sequence` implementation using 64K data buffers and 64K socket queues. The throughput leveled off beyond 64K data buffers.

Unlike the C and ACE wrapper results in Figure 1, the performance of the CORBA versions did not decrease when the size of the data buffers exceeded 8K. 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, however, 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.

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 streaming applications such as teleconferencing and medical imaging [14]. Since the data is untyped the CORBA presentation layer need not perform complex marshalling to handle byte-ordering differences between sender and receiver.

Although marshalling is not required, the CORBA implementations incurred significant data copying overhead. The UNIX execution profiler `prof` was used 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 4 lists the functions where the most time was spent sending and receiving 64 Mbytes using 128K data buffers and 64K 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 128K buffers it was read by the receiver in much smaller chunks of around 8K. 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`).

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 illustrate that the choice of CORBA IDL parameter datatypes has a significant impact on performance. The `sequence` implementations shown in Figure 3 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.

Program	**Socket Queue**	**1 Byte**	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**	**64K**	**128K**
C Code	64K	0.16	23.52	32.19	43.16	61.77	58.10	56.13	48.91	48.6
	8K	0.16	20.77	22.19	23.30	23.57	23.22	22.43	22.63	22.69
ACE Wrapper	64K	0.14	22.29	31.85	42.54	61.81	60.41	56.65	47.67	47.90
	8K	0.14	20.48	22.13	23.66	23.12	23.85	23.58	23.63	23.72
Orbix (Sequence)	64K	0.01	7.85	12.72	18.94	25.79	31.65	34.87	39.15	38.44
	8K	0.01	7.71	11.58	16.60	21.90	22.81	21.89	21.01	20.98
ORBeline (Sequence)	64K	0.01	6.06	12.14	24.36	26.13	35.23	36.70	37.67	36.72
	8K	0.01	10.61	16.09	19.44	21.83	22.75	22.45	21.45	21.05
Orbix (String)	64K	0.01	8.15	13.18	19.02	23.02	29.71	30.98	33.78	32.23
	8k	0.01	8.43	11.62	16.61	22.11	22.76	21.27	20.07	20.00
ORBeline (String)	64K	0.01	9.35	14.97	22.02	22.91	28.99	29.58	28.66	28.42
	8K	0.01	10.22	17.14	20.61	21.16	21.93	19.84	18.28	18.10

Table 1: Results for ATM `ttcp` Tests (in Mbps)

Program	**Socket Queue**	**1 Byte**	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**	**64K**	**128K**
C Code	64K	0.12	8.30	8.46	8.67	8.78	8.66	8.67	8.68	8.71
	8K	0.12	8.10	8.21	8.33	8.30	8.22	8.32	8.36	8.37
ACE Wrapper	64K	0.12	8.22	8.34	8.74	8.72	8.61	8.65	8.68	8.70
	8K	0.12	7.97	8.06	8.38	8.31	8.28	8.19	8.31	8.41
Orbix (Sequence)	64K	0.01	6.68	8.30	8.52	8.51	8.45	8.47	8.44	8.38
	8K	0.01	6.66	7.80	7.97	8.18	8.11	8.20	8.29	8.25
ORBeline (Sequence)	64K	0.01	8.14	8.37	8.63	8.58	8.61	8.64	8.79	8.38
	8K	0.01	7.28	7.70	7.99	8.02	8.21	8.30	8.20	8.22
Orbix (String)	64K	0.01	6.42	8.36	8.55	8.66	8.59	8.58	8.52	8.45
	8k	0.01	6.47	7.82	7.85	8.10	8.17	8.23	8.34	8.30
ORBeline (String)	64K	0.01	8.02	8.44	8.68	8.65	8.67	8.70	8.72	8.29
	8K	0.01	7.40	7.56	7.85	8.00	8.05	8.04	7.99	8.01

Table 2: Results for Ethernet `ttcp` Tests (in Mbps)

The performance variation between Orbix and ORBeline results from differences in their message fragmentation/reassembly implementations, as well as the design of their socket event handling. As shown in Figure 4, 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 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 3).

• **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 does not 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 [15] 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.

2.3 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 qualities of data using TCP/IP over Ethernet and ATM networks. Tables 1 and 2 summarize the results for all the ATM and Ethernet tests, respectively. The remainder of this section evaluates these results and presents recommendations for using DOC frameworks over high-speed networks.

As shown in Table 2, all the benchmark tests perform roughly the same on Ethernet. In contrast, Table 1 illustrates how the data copying overhead of the CORBA implementations significantly limits their throughput on ATM. The performance results illustrate that the overhead of CORBA implementations are not revealed until the network is no longer the limiting factor. In addition, the profiler results in Figure 4 illustrate that small design and implementation differences have a much larger performance impact over high-speed networks than over low-speed networks.

As users and organizations migrate to high-speed networks the performance limitations of contemporary CORBA implementations will become more evident. This should encourage vendors to optimize their ORBs for streaming performance-sensitive applications running over high-speed networks like ATM and FDDI. Key areas of optimization include data copying and data inspection, presentation layer conversions, memory management, and receiver-side demultiplexing and dispatching. In particular, implementations must reduce the number of times that large data buffers are copied on the sender and receiver. The need for these optimizations is widely recognized in the communication protocol community [12] and prototypes that implementate these optimizations are available [16].

Until these optimizations are widely implemented in production systems, however, we recommend that developers of performance-sensitive streaming applications on high-speed networks consider the following when adopting a distributed object computing solution:

- Carefully measure the performance of the communication infrastructure (*i.e.*, the network/host hardware and software). The `ttcp` benchmarks and ACE source code described in this paper are freely available and may be obtained via anonymous ftp from `wuarchive.wustl.edu` in the file `/languages/c++/ACE/` or from URL `http://www.cs.wustl.edu/~schmidt/`. We encourage others to replicate our `ttcp` experiments using different implementations of CORBA and other network/host platforms and report the results.
- Evaluate tools based on empirical measurements and thorough understanding of application requirements, rather than adopting a particular communication model or implementation unconditionally.
- Integrate higher-level DOC frameworks with high-performance object-oriented encapsulations of lower-level network programming interfaces (such as the ACE socket wrappers described in Section 3).
- Insist that CORBA implementors provide hooks to manipulate the underlying protocol layer and socket layer options conveniently. It is particularly important to increase the size of the socket queues to the largest values supported by the OS.
- Tune the size of transmitted data buffers to match the MTU of the network where appropriate.
- Use IDL `sequences` rather than `strings` to avoid unnecessary data access.

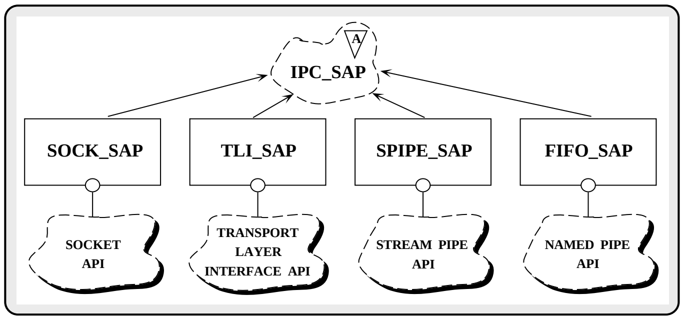

Figure 5: IPC SAP Class Category Relationships

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 [17]. Clearly, implementations of other DOC frameworks (such as OODCE or OLE/COM) that do not address the key sources of overhead on high-speed networks will exhibit similar performance problems.

3 An Object-Oriented Network Programming Interface

Low-level network programming interfaces like sockets or TLI are difficult to program. They require strict attention to many tedious details, making them hard to learn and error prone to use. In addition, programming directly to low-level interfaces limits portability and reuse.

One solution is to develop applications using higher-level distributed object computing (DOC) frameworks like CORBA. DOC frameworks shield developers from low-level programming details and facilitate a reasonably portable distributed computing platform. As described in the previous section, however, the performance of conventional implementations of CORBA may be inadequate for bandwidth-intensive and delay-sensitive streaming applications on high-speed networks.

One method for satisfying the tension between programming simplicity, portability, and run-time efficiency is to encapsulate lower-level network programming interfaces with object-oriented wrappers. Through judicious use of languages features (such as inlining and templates) and design patterns (such as Factories [18], Connectors and Acceptors [19]) it is possible to create reusable object-oriented components that are typesafe, portable, convenient to program, *and* efficient.

This section outlines the design of the `IPC SAP` object-oriented network programming components provided by the ACE toolkit [6]. ACE contains a set of object-oriented networking programming components that perform active and passive connection establishment, data transfer, event demultiplexing, event handler dispatching, routing, dynamic (re)configuration of application services, and concurrency control [6].

`IPC SAP` stands for "InterProcess Communication Service Access Point." It consists of a family of class categories shown in Figure 5 that encapsulate handle-based network programming interfaces such as sockets (`SOCK SAP`), TLI (`TLI SAP`), UNIX SVR4 STREAM pipes (`SPIPE SAP`), and UNIX named pipes (`FIFO SAP`). These network programming wrappers are designed to improve the correctness, programming simplicity, portability, and reusability of performance-sensitive communication software. This section describes the `SOCK SAP` socket wrappers, focusing on design techniques that shield programmers from shortcomings of C, C++, and existing OS network programming interfaces.

3.1 Limitations with Sockets

Sockets were originally developed in BSD UNIX to provide an interface to the TCP/IP protocol suite [4]. From an application's perspective, a socket is a local endpoint of communication that can be bound to an address residing on a local or a remote host. Sockets are accessed via *handles* (also called *descriptors*). Handles are unsigned integers that index into a table maintained in the OS. Handles shield applications from the internal representation of OS data structures. In UNIX and Windows NT, socket handles share the same name space as other handles (such as files, named pipes, and terminal devices).

The standard socket interface is defined by the C functions shown in Figure 6. It contains several dozen routines that perform tasks such as locating address information for network services, establishing and terminating connections, and sending and receiving data [20]. Although the socket interface is widely available and widely used, its design has several notable limitations discussed below. These limitations are shared by other lower-level network programming interfaces such as TLI, STREAM pipes, and named pipes.

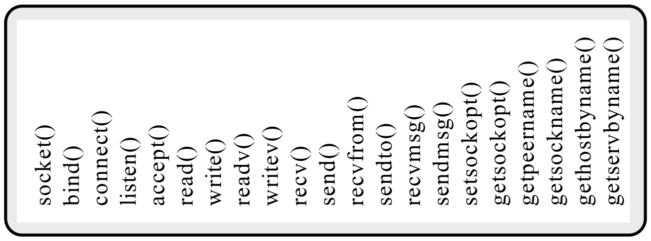

Figure 6: Socket Interface

3.1.1 High Potential for Error

In UNIX any integral value can be passed as a handle to a socket routine. Therefore, compilers are unable to detect or prevent the erroneous use of handles. This weak type-checking allows subtle errors to occur at run-time since the socket interface cannot enforce the correct use of routines for different connection roles (such as active vs. passive connection establishment) or communication services (such as datagram vs. stream communication). Therefore, operations (such as invoking a data transfer operation on a handle designated for establishing connections) can be applied improperly on handles.

Figure 7 depicts the following subtle (and all-to-common) errors that occur when using the socket interface:

1. Forgetting to initialize the `length` parameter (used by `accept`) to the size of `struct sockaddr_in`;
2. Forgetting to "zero-out" all bytes in the socket address structure;
3. Using an address family type that is inconsistent with the protocol family of the socket (*e.g.*, `PF_UNIX` vs. `AF_INET`);
4. Neglecting to use the `htons` library function to convert port numbers from host byte-order to network byte-order and vice versa;
5. Applying the `accept` function on a `SOCK DGRAM` socket;
6. Erroneously omitting parentheses in an assignment expression;
7. Trying to `read` from a passive-mode socket that should only be used to `accept` connections;
8. Failing to properly detect and handle "short-writes" that occur due to buffering in the OS and flow control in the transport protocol.

Other common misuses of sockets not shown in this example are forgetting to call `listen` when creating a passive-mode `SOCK_STREAM` listener socket and miscalculating the length of the pathname in a UNIX-domain socket address (the trailing NUL should not be counted).

Several of the problems listed above are classic problems with programming in C. For instance, by omitting the parentheses in the following expression:

```
int echo_server (u_short port_num)
{
  struct sockaddr_in s_addr;
  int length; // (1) uninitialized variable.
  char buf[BUFSIZ];
  int s_sd, n_sd;
  // Create a local endpoint of communication.
  s_sd = socket (PF_UNIX, SOCK_DGRAM, 0);

  // Set up address information to become a server.
  // (2) forgot to "zero out" structure first...

  // (3) used the wrong address family ...
  s_addr.sin_family = AF_INET;

  // (4) forgot to use htons() on port_num...
  s_addr.sin_port = port_num;
  s_addr.sin_addr.s_addr = INADDR_ANY;

  bind (s_sd, (struct sockaddr *) &s_addr,
        sizeof s_addr) == -1)

  // Create a new endpoint of communication.
  // (5) can't accept() on a SOCK_DGRAM.
  // (6) Omitted a crucial set of parens...
  if (n_sd = accept (s_sd,
                     (struct sockaddr *) &s_addr,
                     &length) == -1) {
    int n;
    // (6) Omitted another set of parens...
    // (7) error to read from s_sd.
    while (n = read (s_sd, buf, sizeof buf) > 0)

      // (8) forgot to check for "short-writes"
      write (n_sd, buf, n);
    // Remainder omitted...
  }
}
```

Figure 7: Socket version of Echo Server

```
if (n_sd = accept (s_sd,
                   (struct sockaddr *) &s_addr,
                   &length) == -1)
```

the value of `n_sd` will always be set to either 0 or 1, depending on whether `accept() == -1`. This problem is exacerbated by the fact that `accept` returns the handle of the newly connected socket. If this handle were passed back as a reference parameter there would be less incentive to use `accept` in an assignment expression.

A deeper problem is that C's lack of support for data abstraction and object-oriented programming makes it hard to define typesafe, reusable, and extensible component interfaces. For example, the generic `sockaddr` socket address structure uses a crude form of inheritance to express the commonality between Internet domain and UNIX domain address structures (`sockaddr_in` and `sockaddr_un`, respectively). These "subclass" address structures require the use of a non-typesafe cast to overlay the `sockaddr` "base class." In an object-oriented language this commonality would be expressed more cleanly and robustly using inheritance and dynamic binding.

In general, the use of unsafe typecasts, combined with the weakly-typed handle-based socket interface, makes it impossible for a compiler to detect mistakes at compile-time. Instead, error checking is deferred until run-time, which complicates error handling and reduces application robustness.

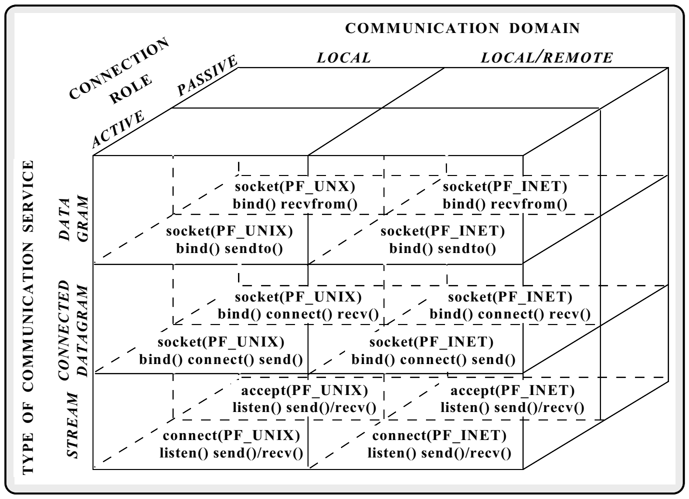
COMMUNICATION DOMAIN
LOCAL
LOCAL/REMOTE
CONNECTION ROLE
PASSIVE
ACTIVE
TYPE OF COMMUNICATION SERVICE
DATA GRAM
CONNECTED DATAGRAM
STREAM
socket(PF_UNX) bind() recvfrom()
socket(PF_INET) bind() recvfrom()
socket(PF_UNIX) bind() sendto()
socket(PF_INET) bind() sendto()
socket(PF_UNIX) bind() connect() recv()
socket(PF_INET) bind() connect() recv()
socket(PF_UNIX) bind() connect() send()
socket(PF_INET) bind() connect() send()
accept(PF_UNIX) listen() send()/recv()
accept(PF_INET) listen() send()/recv()
connect(PF_UNIX) listen() send()/recv()
connect(PF_INET) listen() send()/recv()

Figure 8: A Taxonomy of Socket Communication Dimensions

3.1.2 Complex Interface

Sockets support multiple protocol families (such as TCP/IP, IPX/SPX, ISO OSI, and UNIX domain sockets) with a single interface. The socket interface contains many functions to support different *connection roles* (such as active vs. passive connection establishment), *communication optimizations* (such as `writev`/`readv` that send/receive multiple buffers in a single system call), and *protocol options* (such as broadcasting, multicasting, asynchronous I/O, and urgent data delivery).

Although sockets combine this functionality into a common interface, the result is complex and hard to master. Much of this complexity stems from the overly broad and one-dimensional design of the socket interface. That is, all the routines appear at a single level of abstraction (as shown in Figure 6). This design increases the amount of effort required to learn and use sockets correctly. In particular, programmers must understand most of the interface to use any part of it effectively.

If the socket routines are examined carefully, however, it is clear that the interface decomposes naturally into the following communication dimensions:

1. *Type of communication service – i.e.,* stream vs. datagram vs. connected datagram;
2. *Connection role – i.e.,* active vs. passive (clients are typically active, whereas servers are typically passive);
3. *Communication domain – i.e.,* local IPC only vs. local/remote IPC.

Figure 8 classifies the socket routines according to these dimensions. This natural clustering of functionality is obscured, however, because the socket interface is one-dimensional.

Another problem with the socket interface is that its several dozen routines lack uniform naming conventions. Non-

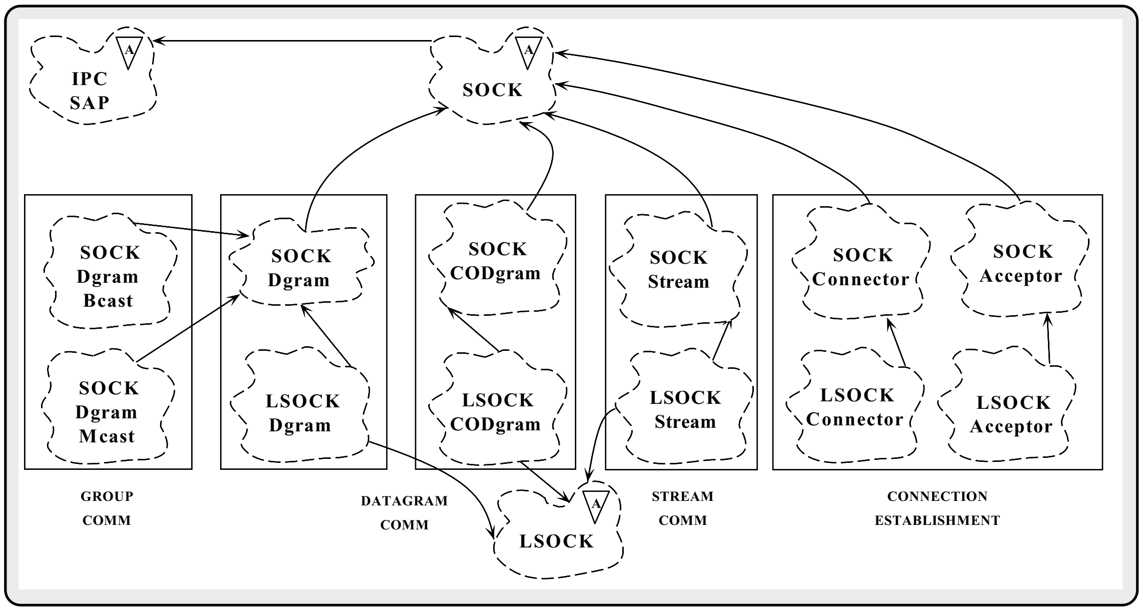

Figure 9: The SOCK SAP Class Categories

uniform naming makes it hard to determine the scope of the socket interface. For example, it is not immediately obvious that `socket`, `bind`, `accept`, and `connect` routines are related. Other network programming interfaces solve this problem by prepending a common prefix before each routine. For example, a "`t_`" is prepended before each routine in the TLI library.

3.2 The SOCK SAP Class Category

`SOCK SAP` is designed to overcome the limitations with sockets described above. It improves the correctness, ease of learning and ease of use, reusability, and portability of communication software. As shown in Section 2, `SOCK SAP` enhances these qualities without sacrificing performance. This section outlines the software architecture of `SOCK SAP` and explains the classes used by the programming examples in Section 4. Readers who are not interested in this level of detail may want to skip to Section 5, which discusses the general principles underlying the design of the `SOCK SAP` wrappers.

`SOCK SAP` consists of around one dozen C++ classes that are related by inheritance and composition. These classes and their relationships are illustrated via Booch notation [21] in Figure 9. Dashed clouds indicate classes and directed edges indicate inheritance relationships between these classes (*e.g.,* `SOCK Stream` inherits from `SOCK`). The general structure of `SOCK SAP` corresponds to the taxonomy of *communication services*, *connection roles*, and *communication domains* shown in Figure 10. It is instructive to compare Figure 8 with Figure 10. The latter is more concise since it uses C++ wrappers to encapsulate the behavior of multiple socket mechanisms within classes related by inheritance.

Each class in `SOCK SAP` provides an abstract interface for a subset of mechanisms that together comprise the overall class category. The functionality of various types of Internet-domain and UNIX-domain sockets is achieved by inheriting mechanisms from the appropriate classes described below. These classes are presented according to the groupings shown in Figure 9.

3.2.1 Base Classes

The `IPC SAP`, `SOCK`, and `LSOCK` classes anchor the inheritance hierarchy and enable subsequent derivation and code sharing. Objects of these classes cannot be instantiated since their constructors are declared in the `protected` section of the class definition.

- **IPC SAP:** This class is the root of the `IPC SAP` hierarchy of C++ wrappers for interprocess communication mechanisms. It provides mechanisms common to all classes, such as handling options like setting a handle into non-blocking mode or enabling asynchronous signal-driven I/O.

- **SOCK:** This class is the root of the `SOCK SAP` hierarchy. It provides mechanisms common to all other classes, such as opening and closing local endpoints of communication and handling options (like selecting socket queue sizes and enabling group communication).

- **LSOCK:** This class provides mechanisms that allow applications to send and receive open file handles between unrelated processes on the local host machine (hence the prefix 'L'). Note that System V and BSD UNIX both support this feature, though Windows NT does not. Other classes inherit from `LSOCK` to obtain this functionality.

`SOCK SAP` distinguishes the `LSOCK*` and `SOCK*` classes on the basis of network address formats and communication

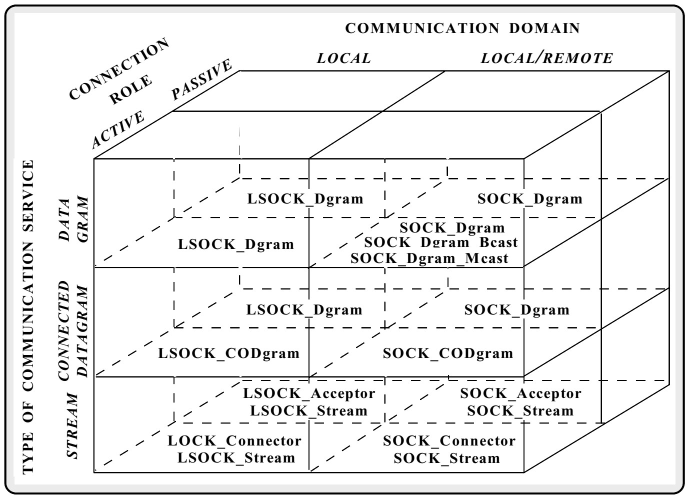

Figure 10: Taxonomy of SOCK SAP Classes and Communication Dimensions

semantics. In particular, the `LSOCK*` classes use UNIX pathnames as addresses and allow only intra-machine IPC. The `SOCK*` classes, on the other hand, use Internet Protocol (IP) addresses and port numbers and allow both intra- and inter-machine IPC.

3.2.2 Connection Establishment

Communication software is typified by asymmetric connection roles between clients and servers. In general, servers listen *passively* for clients to initiate connections *actively* [19]. The structure of passive/active connection establishment and data transfer relationships are captured by the following connection-oriented `SOCK SAP` classes:

- **SOCK Acceptor and LSOCK Acceptor:** The `*Acceptor` classes are factories [18] that passively establish new endpoints of communication in response to active connection requests. The `SOCK Acceptor` and `LSOCK Acceptor` factories produce `SOCK Stream` and `LSOCK Stream` connection endpoint objects, respectively.

- **SOCK Connector and LSOCK Connector:** The `*Connector` classes are factories that actively establish new endpoints of communication. These classes establish connections with remote endpoints and produce the appropriate `*Stream` object when a connection is established. A connection may be initiated either synchronously or asynchronously. The `SOCK Connector` and `LSOCK Connector` factories produce `SOCK Stream` and `LSOCK Stream` connection endpoint objects, respectively.

Note that the `*Acceptor` and `Connector` classes do not provide methods for sending or receiving data. Instead, they are factories that produce the `*Stream` data transfer objects described below. The use of strongly-typed factory interfaces detects and prevents accidental misuse of local and non-local `*Stream` objects at compile-time. In contrast, the socket interface can only detect these type mismatches at run-time.

3.2.3 Stream Communication

Although establishing connections requires a distinction between active and passive roles, once a connection is established data may be exchanged in any order according to the protocol used by the endpoints. `SOCK SAP` isolates the data transfer behavior in the following classes:

- **SOCK Stream and LSOCK Stream:** These two classes are produced by the `*Acceptor` or `*Connector` factories described above. The `*Stream` classes provide mechanisms for transferring data between two processes. `LSOCK Stream` objects exchange data between processes on the same host machine; `SOCK Stream` objects exchange data between processes that can reside on different host machines.

The overloaded `send` and `recv` `*Stream` methods provide standard UNIX `write` and `read` semantics. Thus, a `send` may write less (and a `recv` may read more) than the requested number of bytes. These "short-writes" and "short-reads" occur due to buffering in the OS and flow control in the transport protocol. To reduce programming effort, the the `*Stream` classes provide `send_n` and `recv_n` methods that allow transmission and reception of exactly n bytes. "Scatter-read" and "gather-write" methods are also provided to efficiently send and receive multiple buffers of data simultaneously.

3.2.4 Datagram Communication

This paper has focused primarily on connection-oriented stream communication. However, the socket interface also provides connectionless service that uses the UDP and IP protocols in the Internet protocol suite. UDP and IP are unreliable datagram services that do not guarantee a particular message will arrive at its destination. Connectionless service is used by applications (such as `rwho` daemons [20]) that can tolerate some degree of loss. They also form the foundation for higher-layer reliable protocols.

The `SOCK SAP` socket wrappers encapsulate socket datagram communication with the following classes:

- **SOCK Dgram and LSOCK Dgram:** These classes provide mechanisms for exchanging datagrams between processes running on local and/or remote hosts. Unlike the connected-datagram classes described below, each `send` and `recv` operation must provide the address of the service with every datagram sent or received. `LSOCK Dgram` inherits all the operations of both `SOCK Dgram` and `LSOCK`. It only exchanges datagrams between processes on the same host. The `SOCK Dgram` class, on the other hand, may exchange datagrams between processes on local and/or remote hosts.

• **SOCK CODgram and LSOCK CODgram:** These classes provide a "connected-datagram" mechanism. Unlike the connectionless classes described above, these classes allow the `send` and `recv` operations to omit the address of the service when exchanging datagrams. Note that the connected-datagram mechanism is only a syntactic convenience since there are no additional semantics associated with the data transfer (*i.e.*, datagram delivery remains unreliable). `SOCK CODgram` inherits mechanisms from the `SOCK` base class. `LSOCK CODgram` inherits mechanisms from both `SOCK CODgram` and `LSOCK` (which provides the ability to pass file handles).

3.2.5 Group Communication

Standard TCP and UDP communication is point-to-point. However, some applications benefit from more flexible delivery mechanisms that provide group communication. Therefore, the following classes encapsulate the broadcast and multicast protocols provided by the Internet protocol suite:

• **SOCK Dgram Bcast:** This class provides mechanisms for broadcasting UDP datagrams to processes running on local and/or remote hosts attached to local subnets. The interface for this class supports the broadcast of datagrams to (1) all network interfaces connected to the host machine or (2) a particular network interface. This class shields the end-user from the low-level details required to utilize broadcasting effectively.

• **SOCK Dgram Mcast:** This class provides mechanisms for multicasting UDP datagrams to processes running on local and/or remote hosts attached to local subnets. The interface for this class supports the multicast of datagrams to a particular multicast group. This class shields the end-user from the low-level details required to utilize multicasting effectively.

3.3 Network Addressing

Designing an efficient, general-purpose network addressing interface is hard. The difficulty stems from trying to represent different network address formats with a space efficient and uniform interface. Different address formats store diverse types of information represented with various sizes. For example, an Internet-domain service (such as `ftp` or `telnet`) is identified using two fields: (1) a four-byte IP address (which uniquely identifies the remote host machine throughout the Internet) and (2) a two-byte port number (which is used to demultiplex incoming protocol data units to the appropriate client or server process on the remote host machine). In contrast, UNIX-domain sockets rendezvous via UNIX pathnames (which may be up to 108 bytes in length and are meaningful only on a single local host machine).

The existing `sockaddr`-based network addressing structures provided by the socket interface is cumbersome and error-prone. It requires developers to explicitly initialize all the bytes in the address structure to 0 and to use explicit casts. In contrast, the `SOCK SAP` addressing classes shown in Figure 11 contain mechanisms for manipulating network addresses. The constructors for the `Addr` base class ensure that all fields are automatically initialized correctly. Moreover, the different sizes, formats, and functionality that exist between different address families are encapsulated in the derived address subclasses. This makes it easier to extend the network addressing scheme to encompass new communication domains. For example, the `UNIX Addr` subclass is associated with the `LSOCK*` classes, the `INET Addr` subclass is associated with the `SOCK*` and `TLI*` classes, and the `SPIPE Addr` subclass is associated with the STREAM pipe wrappers in `SPIPE SAP`.

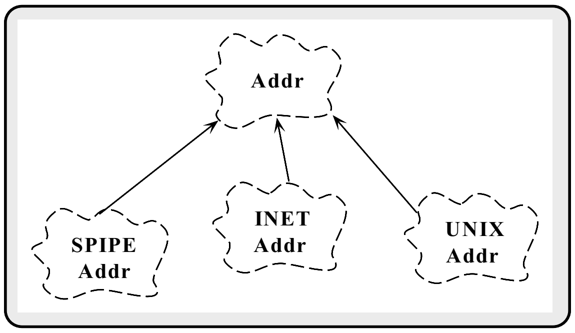

Figure 11: The `SOCK SAP` Address Class Hierarchy

4 Programming with SOCK SAP C++ Wrappers

This section illustrates the ACE `SOCK SAP` C++ wrappers by using them to develop a client/server streaming application. This application is simplified version of the `ttcp` program described in Section 2. For comparison, this application is also written with sockets and CORBA. Most of the error checking has been omitted in these examples to keep them short. Naturally, robust programs should check the return values of library and system calls.

Figures 12 and 13 present a client/server program written in C that uses Internet-domain sockets and `select` to implement the stream application. The server shown in Figure 13 creates a passive-mode listener socket and waits for clients to connect to it. Once connected, the server receives the data transmitted from the client and displays the data on its standard output stream. The client-side shown in Figure 12 establishes a TCP connection with the server and transmits its standard input stream across the connection. The client uses non-blocking connections to limit the amount of time it waits for a connection to be accepted or refused.

Most of the error checking for return values has been omitted to save space. However, it is instructive to note all the socket initialization, network addressing, and flow control details that must be programmed explicitly to make even this simple example work correctly. Moreover, the code in Figures 12 and 13 is not portable to platforms that do not support both sockets and `select`.

Figures 14 and 15 use `SOCK SAP` to reimplement the C versions of the client/server programs. The `SOCK SAP` programs implement the same functionality as those presented in Figure 12 and Figure 13. The `SOCK SAP` C++ programs exhibit the following benefits compared with the socket-based C implementation:

- *Increased clarity – e.g.*, network addressing and host location is handled by the `Addr` class shown in Figure 11, which hides the subtle and error-prone details that must be programmed explicitly in Figures 12 and 13. Moreover, the low-level details of non-blocking connection establishment are performed by the `SOCK Connector` factory.
- *Increased typesafety – e.g.*, the `SOCK Acceptor` and `SOCK Connector` connection factories create `SOCK Stream` objects. This prevents the type errors shown in Figure 7 from occurring at run-time.
- *Decreased program size – e.g.*, a substantial reduction in the lines of code results from localizing active and passive connection establishment in the `SOCK Acceptor` and `SOCK Connector` connection factories. In addition, default values are provided for constructor and method parameters, which reduces the number of arguments needed for common usage patterns.
- *Increased portability – e.g.*, switching between sockets and TLI simply requires changing

  ```
  send_data <TLI_Connector, TLI_Stream,
              INET_Addr> (s_addr);
  ```

  in the client to

  ```
  send_data <SOCK_Connector, SOCK_Stream,
              INET_Addr> (s_addr);
  ```

  and

  ```
  recv_data<SOCK_Acceptor, SOCK_Stream,
             INET_Addr> (s_addr);
  ```

  in the server to

  ```
  recv_data<TLI_Acceptor, TLI_Stream,
             INET_Addr> (s_addr);
  ```

  Conditional compilation directives can be used to further decouple the communication software from reliance upon a particular type of network programming interface.

However, the ACE wrappers share some of the same drawbacks as sockets. In particular, too much of the code required to program at this level is not directly related to the application. In contrast, Figures 16 and 17 illustrate the CORBA version of the stream application implemented using Orbix 1.3. This implementation is more concise than both the C and ACE C++ wrapper versions. CORBA performs the low-level communication details associated with service location, passive and active connection establishment, message framing, marshalling and demarshalling, demultiplexing, and upcall

```
#define PORT_NUM 10000
#define TIMEOUT 5

/* Socket client. */

void send_data (const char host[], u_short port_num)
{
  struct sockaddr_in peer_addr;
  struct hostent *hp;
  char buf[BUFSIZ];
  int s_sd, w_bytes, r_bytes, n;

  /* Create a local endpoint of communication */
  s_sd = socket (PF_INET, SOCK_STREAM, 0);

  /* Set s_sd to non-blocking mode. */
  n = fcntl (s_sd, F_GETFL, 0);
  fcntl (s_sd, F_SETFL, n | O_NONBLOCK);

  /* Determine IP address of the server */
  hp = gethostbyname (host);

  /* Set up address information to contact server */
  memset ((void *) &peer_addr, 0, sizeof peer_addr);
  peer_addr.sin_family = AF_INET;
  peer_addr.sin_port = port_num;
  memcpy (&peer_addr.sin_addr,
          hp->h_addr, hp->h_length);

  /* Establish non-blocking connection server. */
  if (connect (s_sd, (struct sockaddr *) &peer_addr,
               sizeof peer_addr) == -1) {
    if (errno == EINPROGRESS) {
      struct timeval tv = {TIMEOUT, 0};
      fd_set rd_sds, wr_sds;
      FD_ZERO (&rd_sds);
      FD_ZERO (&wr_sds);
      FD_SET (s_sd, &wr_sds);
      FD_SET (s_sd, &rd_sds);

      /* Wait up to TIMEOUT seconds to connect. */
      if (select (s_sd + 1, &rd_sds, &wr_sds,
                  0, &tv) <= 0)
        perror ("connection timedout"), exit (1);
      // Recheck if connection is established.
      if (connect (s_sd,
                   (struct sockaddr *) &peer_addr,
                   sizeof peer_addr) == -1
          && errno != EISCONN)
        perror ("connect failed"), exit (1);
    }
  }

  /* Send data to server (correctly handles
     "short writes" due to flow control) */

  while ((r_bytes = read (0, buf, sizeof buf)) > 0)
    for (w_bytes = 0; w_bytes < r_bytes; w_bytes += n)
      n = write (s_sd, buf + w_bytes,
                 r_bytes - w_bytes);

  /* Close down the connection. */
  close (s_sd);
}

int main (int argc, char *argv[])
{
  char *host = argc > 1 ? argv[1] : "ics.uci.edu";
  u_short port_num =
    htons (argc > 2 ? atoi (argv[2]) : PORT_NUM);

  /* Send data to the server. */
  send_data (host, port_num);
  return 0;
}
```

Figure 12: Socket-based Client Example

```
#define PORT_NUM 10000

/* Socket server. */

void recv_data (u_short port_num)
{
  struct sockaddr_in s_addr;
  int s_sd;

  /* Create a local endpoint of communication */
  s_sd = socket (PF_INET, SOCK_STREAM, 0);

  /* Set up the address information for a server */
  memset ((void *) &s_addr, 0, sizeof s_addr);
  s_addr.sin_family = AF_INET;
  s_addr.sin_port = port_num;
  s_addr.sin_addr.s_addr = INADDR_ANY;

  /* Associate address with endpoint */
  bind (s_sd, (struct sockaddr *) &s_addr,
        sizeof s_addr);

  /* Make endpoint listen for service requests */
  listen (s_sd, 5);

  /* Performs the iterative server activities */

  for (;;) {
    char buf[BUFSIZ];
    int r_bytes, n_sd;
    struct sockaddr_in peer_addr;
    int peer_addr_len = sizeof peer_addr;
    struct hostent *hp;

    /* Create a new endpoint of communication */
    while ((n_sd = accept (s_sd, &peer_addr,
                           &peer_addr_len)) == -1
           && errno == EINTR)
      continue;

    hp = gethostbyaddr (&peer_addr.sin_addr,
                        peer_addr_len, AF_INET);

    printf ("client %s\n", hp->h_name);

    /* Read data from client (terminate on error) */

    while ((r_bytes = read (n_sd, buf, sizeof buf)) > 0)
      write (1, buf, r_bytes);

    /* Close the new endpoint
       (listening endpoint remains open) */
    close (n_sd);
  }
  /* NOTREACHED */
}

int main (int argc, char *argv[])
{
  u_short port_num =
    htons (argc > 1 ? atoi (argv[1]) : PORT_NUM);

  // Receive data from clients.
  recv_data (port_num);
  return 0;
}
```

Figure 13: Socket-based Server Example

```
static const int PORT_NUM = 10000;
static const int TIMEOUT = 5;

// SOCK_SAP Client.

template <class CONNECTOR,
          class STREAM,
          class ADDR>
void send_data (ADDR peer_addr)
{
  // Data transfer object.
  STREAM peer_stream;

  // Establish connection without blocking.
  CONNECTOR connector
    (peer_stream, peer_addr, ACE_NONBLOCK);

  if (peer_stream.get_handle () == -1) {
    // If non-blocking connection is in progress,
    // wait up to TIMEOUT seconds to complete.
    Time_Value timeout (TIMEOUT);

    if (errno != EWOULDBLOCK ||
        connector.complete
         (peer_stream, peer_addr, &timeout) == -1)
      perror ("connector"), exit (1);
  }

  // Send data to server (send_n() handles
  // "short writes" correctly).

  char buf[BUFSIZ];

  for (int r_bytes;
       (r_bytes = read (0, buf, sizeof buf)) > 0;)
    peer_stream.send_n (buf, r_bytes);

  // Explicitly close the connection.
  peer_stream.close ();
}

int main (int argc, char *argv[])
{
  char *host = argc > 1 ? argv[1] : "ics.uci.edu";
  u_short port_num =
    htons (argc > 2 ? atoi (argv[2]) : PORT_NUM);

  // Address of the server.
  INET_Addr s_addr (port_num, host)

  // Use TLI wrappers on client's side.
  send_data <TLI_Connector, TLI_Stream,
             INET_Addr> (s_addr);
  return 0;
}
```

Figure 14: SOCK SAP-based Client Example

```
static const int PORT_NUM = 10000;

// SOCK_SAP Server.

template <class ACCEPTOR,
          class STREAM,
          class ADDR>
void recv_data (ADDR s_addr)
{
  // Factory for passive connection establishment.
  ACCEPTOR acceptor (s_addr);

  // Data transfer object.
  STREAM peer_stream;

  // Remote peer address.
  ADDR peer_addr;

  // Performs iterative server activities.

  for (;;) {
    // Create a new STREAM endpoint
    // (automatically restarted if errno == EINTR).
    acceptor.accept (peer_stream, &peer_addr);

    printf ("client %s\n", peer_addr.get_host_name ());

    // Read data from client (terminate on error).

    char buf[BUFSIZ];

    for (int r_bytes;
         peer_stream.recv (buf, sizeof buf,
                           r_bytes) > 0;)
      write (1, buf, r_bytes);

    // Close peer_stream endpoint
    // (acceptor endpoint stays open).
    peer_stream.close ();
  }
  /* NOTREACHED */
}

int main (int argc, char *argv[])
{
  u_short port_num =
    argc == 1 ? PORT_NUM : atoi (argv[1]);

  // Port for the server.
  INET_Addr s_addr (port_num);

  // Use socket wrappers on server's side.
  recv_data<SOCK_Acceptor, SOCK_Stream,
            INET_Addr> (s_addr);
  return 0;
}
```

Figure 15: SOCK SAP-based Server Example

```
// CORBA IDL interface.

interface Data_Stream
{
  typedef sequence<char> Stream_Buf;

  exception Disconnected {};

  oneway void send (in Stream_Buf buf)
    raises (Disconnected);
};

// CORBA Client.

void send_data (Data_Stream *peer_stream)
{
  // Constructor allocates memory.
  Data_Stream::Stream_Buf buf (BUFSIZ);

  // Read from stdin and send to server.

  while ((buf._length =
          read (0, buf._buffer, BUFSIZ)) > 0)
    peer_stream->send (buf);

  // Decrement object reference's ref count.
  peer_stream->_release ();
}

int main (int argc, char *argv[])
{
  char *marker
    = argc > 1 ? argv[1] : "data_stream";
  Data_Stream *peer_stream =
    Data_Stream::_bind (marker);

  send_data (peer_stream);
  return 0;
}
```

Figure 16: CORBA-based Client Example

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

class Data_Stream_i
  : public Data_StreamBOAImpl
{
  // Upcall invoked by the CORBA skeleton.
  virtual void send
    (const Data_Stream::Stream_Buf &,
     CORBA::Environment &);
};

// Upcall invoked by the CORBA skeleton.

void
Data_Stream_i::send
  (const Data_Stream::Stream_Buf &buf,
   CORBA::Environment &IT_env)
{
  // Write data to standard output.
  write (1, buf._buffer, buf._length);
}

// CORBA persistent server.

int main (int argc, char *argv[])
{
  char *executable = argv[0];
  char *marker
    = argc > 1 ? argv[1] : "data_stream";

  // Define an implementation object.
  Data_Stream_i data_stream (marker);

  // ACE method that registers service
  // with the ORB automatically.
  CORBA_Handler::activate_service
    ("Data_Stream", object_name, executable);

  // Tell the ORB that the objects are active.
  CORBA::Orbix.impl_is_ready ("Data_Stream");

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

Figure 17: CORBA-based Server Example

dispatching. This allows developers to concentrate on defining application-specific behavior, rather than wrestling with the details of network programming.

The persistent CORBA server shown in Figure 17 creates an implementation of a `Data_Stream` IDL interface and informs the ORB that it is ready to receive `send` requests from clients. It uses a standard ACE class `CORBA_Handler` to automatically register the server and object name with the Orbix daemon.

The client shown in Figure 16 uses the Orbix locator service to bind to the `marker` exported by the `Data_Stream` server. Once bound, the client transmits all data from its standard input to the server via the `Data_Stream::send` proxy. This example behaves slightly differently than the C and ACE wrapper versions since CORBA does not provide a standard means to obtain the host and port of the sender. Moreover, CORBA communication semantics are request-oriented rather than connection-oriented. Thus, other clients could conceivably bind to the same marker name and transmit data via its `send` method. To get the same behavior with CORBA would require the use of an object factory.

5 Socket Wrapper Design Principles

This section describes the following design principles that are applied throughout the `SOCK SAP` class category:

- Enforce typesafety at compile-time
- Allow controlled violations of typesafety
- Simplify for the common case
- Replace one-dimensional interfaces with hierarchical class categories
- Enhance portability with parameterized types
- Inline performance critical methods
- Define auxiliary classes to hide error-prone details

Although these principles are widely known and widely used in domains like graphical user interfaces they have been less widely applied in the communication software domain.

• **Enforce typesafety at compile-time:** Several limitations with sockets discussed in Section 3.1 stem from the lack of typesafety in its interface. To enforce typesafety, `SOCK SAP` ensures all its objects are properly initialized via constructors. In addition, to prevent accidental violations of typesafety, only legal operations are permitted on `SOCK SAP` objects. This latter point is illustrated in the `SOCK SAP` revision of `echo_server` shown in Figure 18. This version fixes the problems with sockets and C identified in Figure 7. Since `SOCK SAP` classes are strongly typed, invalid operations are rejected at compile-time rather than at run-time. For example, it is not possible to invoke `recv` or `send` on a `SOCK_Acceptor` connection factory since these methods are not part of its interface. Likewise, return values are only used to convey success or failure of operations. This reduces the potential for misuse in assignment expressions.

```
int echo_server (u_short port_num)
{
  // Address of local server.
  INET_Addr s_addr (port_num);

  // Initialize the passive mode server.
  SOCK_Acceptor acceptor (s_addr);

  // Data transfer object.
  SOCK_Stream peer_stream;

  // Client remote address object.
  INET_Addr peer_addr;

  // Accept a new connection.
  if (acceptor.accept (peer_stream,
                       &peer_addr) != -1) {
    char buf[BUFSIZ];
    for (size_t n;
         peer_stream.recv (buf, sizeof buf, n) > 0;)
      // Handles "short-writes."
      if (peer_stream.send_n (buf, n) != n)
        // Remainder omitted.
  }
}
```

Figure 18: SOCK SAP Revision of the Echo Server

• **Allow controlled violations of typesafety:** This principle is exemplified by the get_handle and set_handle methods provided by the IPC SAP root class. These methods extract and assign the underlying handle, respectively. By providing get_handle and set_handle, IPC SAP allows applications to circumvent its type-checking mechanisms in situations where applications must interface directly with UNIX system calls (such as select) that expect a handle. Another way of stating this principle is "make it easy to use SOCK SAP correctly, hard to use it incorrectly, but not impossible to use it in ways the class designers did not anticipate."

• **Simplify for the common case:** This principle is applied in the following ways in the ACE C++ socket wrappers:

- *Supply default parameters for common method arguments* – for instance, the SOCK Connector constructor has six parameters:

```
SOCK_Connector
  (SOCK_Stream &new_stream,
   const Addr &remote_sap,
   int blocking_semantics = 0,
   const Addr &local_sap = Addr::sap_any,
   int protcol_family = PF_INET,
   int protocol = 0);
```

  However, only the first two commonly vary from call to call:

```
SOCK_Stream stream;
// Compiler supplies default values.
SOCK_Connector con (stream,
                    INET_Addr (port, host));
// ...
```

  Therefore, to simplify programming, the values are given as defaults in the SOCK Connector constructor so that programmers need not provide them every time.

- *Define parsimonious interfaces* – This principle localizes the cost of using a particular abstraction. The IPC SAP interfaces limits the amount of details that application developers must remember. IPC SAP provides developers with clusters of classes that perform various types of communication (such as connection-oriented vs. connectionless) and various connection roles (such as active vs. passive). To reduce the chance of error, the SOCK Acceptor class only permits operations that apply for programs playing passive roles and the SOCK Connector class only permits operations that apply for programs playing an active role. In addition, sending and receiving open file handles has a much simpler calling interface using SOCK SAP compared with using the highly-general UNIX sendmsg/recvmsg routines. For example, using LSOCK* classes to pass socket descriptors is very concise:

```
LSOCK_Stream stream;
LSOCK_Acceptor acceptor ("/tmp/foo");

// Accept connection.
acceptor.accept (stream);

// Pass the socket descriptor back to caller.
stream.send_handle (stream.get_handle ());
```

  versus the code that is required to implement this using the socket interface:

```
int n_sd;
int u_sd;
sockaddr_un addr;
u_char a[2];
iovec  iov;
msghdr send_msg;

u_sd = socket (PF_UNIX, SOCK_STREAM, 0);

memset ((void *) &addr, 0, sizeof addr);
addr.sun_family = AF_UNIX;
strcpy (addr.sun_path, "/tmp/foo");

bind (u_sd, &addr, sizeof addr.sun_family +
                   strlen ("/tmp/foo"));
listen (u_sd, 5);

// Accept connection.
n_sd = accept (u_sd, 0, 0);

// Sanity check.
a[0] = 0xab; a[1] = 0xcd;

iov.iov_base = (char *) a;
iov.iov_len = sizeof a;

send_msg.msg_iov = &iov;
send_msg.msg_iovlen = 1;
send_msg.msg_name = (char *) 0;
send_msg.msg_namelen = 0;
send_msg.msg_accrights = (char *) &n_sd;
send_msg.msg_accrightslen = sizeof n_sd;

// Pass the socket descriptor back to caller.
sendmsg (n_sd, &send_msg, 0);
```

- *Combine multiple operations into a single operation* – Creating a conventional passive-mode socket requires multiple calls:

```
int s_sd = socket (PF_INET, SOCK_STREAM, 0);
sockaddr_in addr;
memset (&addr, 0, sizeof addr);
addr.sin_family = AF_INET;
addr.sin_port = htons (port);
addr.sin_addr.s_addr = INADDR_ANY;
```

```
template <class ACCEPTOR,
          class STREAM,
          class ADDR>
int echo_server (u_short port_num)
{
  // Local address of server.
  ADDR s_addr (port_num);

  // Initialize the passive mode server.
  ACCEPTOR acceptor (s_addr);

  // Data transfer object.
  STREAM peer_stream;

  // Remote address object.
  ADDR peer_addr;

  // Accept a new connection.
  if (acceptor.accept (peer_stream,
                       &peer_addr) != -1) {
    char buf[BUFSIZ];
    for (size_t n;
         peer_stream.recv (buf, sizeof buf,
                           n) > 0;)
      if (peer_stream.send_n (buf, n) != n)
        // Remainder omitted.
  }
}
```

Figure 19: Template Version of the Echo Server

```
bind (s_sd, &addr, addr_len);
listen (s_sd);
// ...
```

In contrast, the SOCK Acceptor is a factory for passive connection establishment. Its constructor performs the socket calls socket, bind, and listen required to create a passive-mode listener endpoint. Therefore, applications simply write the following:

```
SOCK_Acceptor acceptor (INET_Addr (port));
```

to achieve the functionality presented above.

• **Replace one-dimensional interfaces with hierarchical class categories:** This principle involves using hierarchically-related class categories to restructure existing one-dimensional socket interfaces (shown in Figure 9). The criteria used to structure the SOCK SAP class category involved identifying, clustering, and encapsulating related socket routines to maximize the reuse and sharing of class components.

Inheritance supports different subsets of functionality for the SOCK SAP class categories. For instance, not all operating systems support passing open file handles (*e.g.,* Windows NT). Thus, it is possible to omit the LSOCK class (described in Section 3.2) from the inheritance hierarchy without affecting the interfaces of other classes in the SOCK SAP design.

Inheritance also increases code reuse and improves modularity. Base classes express *similarities* between class category components and derived classes express the *differences*. For example, the IPC SAP design places shared mechanisms towards the "root" of the inheritance hierarchy in the IPC SAP and SOCK SAP base classes. These mechanisms include operations for opening/closing and setting/retrieving the underlying socket handles, as well as certain option management functions that are common to all the derived SOCK SAP classes. Subclasses located towards the "bottom" of the inheritance hierarchy implement specialized operations that are customized for the type of communication provided (such as stream vs. datagram communication or local vs. remote communication). This approach avoids unnecessary duplication of code since the more specialized derived classes reuse the more general mechanisms provided at the root of the inheritance hierarchy.

• **Enhance portability with parameterized types:** Wrapping sockets with C++ classes (rather than stand-alone C functions) helps to improve portability by allowing the wholesale replacement of network programming interfaces via parameterized types. Parameterized types decouple applications from reliance on specific network programming interfaces. Figure 19 illustrates this technique by modifying the echo_server to become a C++ function template. Depending on certain properties of the underlying OS platform (such as whether it implements TLI or sockets more efficiently), the echo_server may be instantiated with either SOCK SAP or TLI SAP classes, as shown below:

```
// Conditionally select IPC mechanism.
#if defined (USE_SOCKETS)
typedef SOCK_Stream STREAM;
typedef SOCK_Acceptor ACCEPTOR;
#else
typedef TLI_Stream STREAM;
typedef TLI_Acceptor ACCEPTOR;
#endif // USE_SOCKETS.

const int PORT_NUM = 10000;

int main (void)
{
  // ...

  // Invoke the echo_server with appropriate
  // network programming interfaces.
  echo_server<ACCEPTOR, STREAM,
              INET_Addr> (PORT_NUM);
}
```

In general, the use of parameterized types is less intrusive and more extensible that conventional alternatives (such as implementing multiple versions or littering conditional compilation directives throughout the source code).

• **Inline performance critical methods:** To encourage developers to replace existing low-level network programming interfaces with C++ wrappers, the SOCK SAP implementation must operate efficiently. To ensure this, methods in the critical performance path (such as the SOCK Stream recv and send methods) are specified as C++ inline functions to eliminate run-time overhead. Inlining is both time and space efficient since these methods are very short (approximately 2 or 3 lines per method). The use of inlining implies that virtual functions should be used sparingly since most contemporary C++ compilers do not fully optimize away virtual function overhead.

• **Define auxiliary classes that hide error-prone details:** The C interface to socket addressing is awkward and error-prone. It is easy to neglect to zero-out a `sockaddr_in` or convert port numbers to network byte-order. To shield applications from these low-level details, `IPC SAP` define the `Addr` class hierarchy (shown in Figure 11). This hierarchy supports several diverse network addressing formats via a typesafe C++ interface. The `Addr` hierarchy eliminates common programming errors associated with using the C-based family of `struct sockaddr` data structures directly. For example, the constructor of `INET Addr` automatically zeros-out the `sockaddr` addressing structure and converts the port number to network byte order, as follows:

```
class INET_Addr : public Addr {
public:
  INET_Addr::INET_Addr (u_short port,
                        long ip_addr = 0) {
    memset (&this->inet_addr_, 0,
            sizeof this->inet_addr_);
    this->inet_addr_.sin_family = AF_INET;
    this->inet_addr_.sin_port = htons (port);
    memcpy (&this->inet_addr_.sin_addr,
            &ip_addr, sizeof ip_addr);
  }
private:
  sockaddr_in inet_addr_;
};
```

6 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 ACE socket wrappers described in this paper provide a high-performance network programming interface that shields developers from lower-level details of sockets or TLI without sacrificing performance. The ACE wrappers automate and simplify many aspects (such as initialization, addressing, and handling short-writes) of using lower-level network programming interfaces. They improve portability by shielding applications from platform-specific network programming interfaces. Wrapping sockets with C++ classes (rather than stand-alone C functions) makes it convenient to switch wholesale between different network programming interfaces by using parameterized types. In addition, as shown in Figure 2, the ACE socket wrappers do not introduce any significant overhead compared with programming to sockets directly.

The primary drawback with the ACE network programming wrappers is that they do not address higher-level issues related to system reliability and availability, flexibility of object location and selection, support for transactions, security, and deferred process activation, and the exchange of binary data between different computer architectures. For example, programmers must explicitly program presentation layer conversions in conjunction with the ACE wrappers. Therefore, these wrappers are most useful when the datatypes are simple (like those used by the high-performance streaming applications described in this paper).

The ACE C++ wrappers for sockets may be integrated with CORBA to enhance the performance of streaming applications. We've combined CORBA and the ACE wrappers in a high-speed teleradiology system that transfers 10-40 Mbyte medical images over ATM. In this system, CORBA is used as a signaling mechanism to identify endpoints of communication in a location-independent manner. The ACE 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.

ACE has been ported to many versions of UNIX and Windows NT and is currently being used in many commercial products including the Bellcore and Siemens Q.port ATM signaling software product, the Ericsson EOS family of telecommunication monitoring applications, the System Control Segment of the Motorola Iridium project, and a high-speed enterprise-wide medical image delivery system for Kodak Health Imaging Systems.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1.2 ed., 1993.

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

[5] Sun Microsystems, *Network Interfaces Programmer's Guide*, Chapter 6 (TLI Interface) ed., 1992.

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

[7] D. C. Schmidt, "A System of Reusable Design Patterns for Application-level Gateways," in *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)* (S. P. Berczuk, ed.), Wiley and Sons, 1995.

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

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

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

[11] D. C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communications Systems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 489–506, May 1993.

[12] D. D. Clark and D. L. Tennenhouse, “Architectural Considerations for a New Generation of Protocols,” in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 200–208, ACM, Sept. 1990.

[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), pp. 176–185, IEEE, Oct. 1994.

[14] M. DoVan, L. Humphrey, G. Cox, and C. Ravin, “Initial Experience with Asynchronous Transfer Mode for Use in a Medical Imaging Network,” *Journal of Digital Imaging*, vol. 8, pp. 43–48, February 1995.

[15] D. C. Schmidt, “GPERF: A Perfect Hash Function Generator,” in *Proceedings of the* 2^{nd} *C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.

[16] S. W. O’Malley, T. A. Proebsting, and A. B. Montz, “Universal Stub Compiler,” in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, Aug. 1994.

[17] K. Birman and R. van Renesse, “RPC Considered Inadequate,” in *Reliable Distributed Computing with the Isis Toolkit*, pp. 68–78, Los Alamitos: IEEE Computer Society Press, 1994.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

[19] D. C. Schmidt, “Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services,” in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP ’95*, (Aarhus, Denmark), August 1995.

[20] W. R. Stevens, *UNIX Network Programming*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[21] G. Booch, *Object Oriented Analysis and Design with Applications (*2^{nd} *Edition)*. Redwood City, California: Benjamin/Cummings, 1993.