IPC SAP

A Family of Object-Oriented Interfaces for Local and Remote Interprocess Communication

Douglas C. Schmidt
schmidt@cs.wustl.edu
http://www.cs.wustl.edu/~schmidt/
Department of Computer Science
Washington University, St. Louis 63130
(TEL) 314-935-7538, (FAX) 314-935-7302

An earlier version of this paper appeared in the November/December 1992 issue of the C++ Report magazine. An expanded version of this paper [1] that includes performance results over Ethernet and ATM networks is available at the WWW URL `http://www.cs.wustl.edu/~schmidt/COOTS-95.ps.Z`.

1 Introduction

This is part of a series that describes techniques for encapsulating existing operating system (OS) interprocess communication (IPC) mechanisms within object-oriented C++ wrappers. This paper focuses on the `IPC SAP` ("Inter-Process Communication Service Access Point") C++ class categories. `IPC SAP` provides applications with a family of object-oriented service access points that encapsulate standard network programming interfaces such as BSD sockets [2], the System V transport layer interface (TLI) [3], SVR4 STREAM pipes [4], UNIX FIFOs [5], and Windows NT named pipes [6]. In turn, these network programming interfaces shield developers and applications from non-portable details of the underlying OS local and remote IPC mechanisms. IPC mechanisms include standard connection-oriented and connectionless protocols (such as TCP, UDP, and IPX/SPX) available in modern operating systems (such as BSD/SVR4 UNIX, Windows 3.1, Win32, and OS/2). `IPC SAP` utilizes object-oriented techniques and C++ features to provide a rich set of components that simplify the development of communication software.

This paper is organized as follows: Section 2 outlines the levels of abstraction for programming communication software; Section 3 describes existing network programming interfaces and discusses their limitations; Section 4 presents the object-oriented design and implementation of `IPC SAP` and explains how it overcomes limitations with existing interfaces; Section 5 examines the object-oriented design of the C++ wrappers for sockets, TLI, STREAM pipes, and FIFOs in detail; Section 6 illustrates several examples that use `IPC SAP` to implement a client/server application; Section 7 discusses the principles the guided the design of `IPC SAP`; and Section 8 summarizes the advantages and disadvantages of using C++ to develop object-oriented wrappers.

2 Network Programming Interfaces

Writing robust, extensible, and efficient communication software is a complex and challenging task. Developers must master many complex OS and communication concepts such as:

- Network addressing and service identification;
- Presentation conversions (*e.g.,* encryption, compression, and network byte-ordering conversions between heterogeneous end-systems with alternative processor byte-orderings);
- Process and thread creation and synchronization;
- System call and library routine interfaces to local and remote interprocess communication (IPC) mechanisms.

Various programming tools and interfaces have been created to help simplify the development of communication software. Figure 1 illustrates the IPC interfaces available on modern OS platforms (such as SVR4 UNIX). As shown in the figure, applications may access several levels of network programming interfaces for local and remote IPC. The remainder of this section outlines each level of abstraction, ranging from high-level RPC toolkits to transport layer interfaces to low-level kernel programming mechanisms.

2.1 RPC Toolkits

Applications that exchange data with clients in a "request-response" fashion (such as network file servers) are often developed using code produced by remote procedure call (RPC) toolkits [7]. RPC toolkits provide a set of high-level tools (such stub generators and directory services) that support request-response and "oneway" communication. RPC toolkits are defined broadly to include the Open Network Computing (ONC+) transport-independent RPC facility [7], the Common Object Request Broker Architecture (CORBA) [8], the OSF Distributed Computing Environment (DCE)[1] [9] and OODCE [10], and Microsoft's OLE/COM [11]. RPC toolkits automate many tedious and error-prone aspects of these application development including

[1]DCE also provides support for bulk data transfer via its *pipe* datatype, which is not provided by CORBA or ONC RPC.

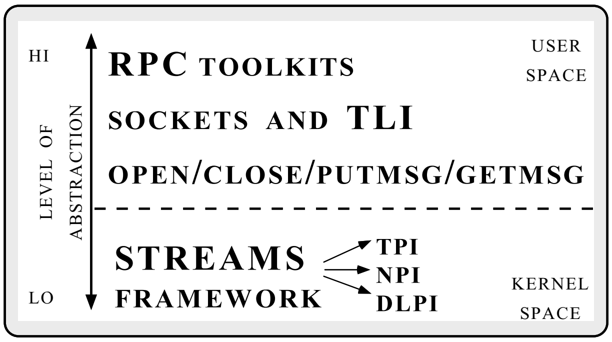

Figure 1: Levels of Abstraction for Network Programming

- Authentication, authorization, and data security;
- Service location and binding;
- Service registration and activation;
- Demultiplexing and dispatching in response to events;
- Implementing message framing atop bytestream-oriented communication protocols like TCP;
- Presentation conversion issues involving network byte-ordering and parameter marshaling.

In addition, RPC toolkits shield developers from the complexities of lower-level OS system calls that transmit and receive packets across a network.

2.2 Transport Layer Interfaces

RPC toolkits are typically built upon transport layer interfaces such as BSD UNIX sockets [2], the System V UNIX Transport Layer Interface (TLI) [3], or Windows NT named pipes [6]. Compared with the higher-level RPC toolkits, there are several advantages to directly programming applications via transport layer interfaces:

- *Omit unnecessary functionality* – Applications may omit unnecessary functionality (such as presentation layer conversions for ASCII data);
- *Allow fine-grained control over behavior* – Transport interfaces enable finer-grain control over behavior (such as permitting multicast transmission and signal-driven asynchronous I/O);
- *Increase Portability* – Transport interface (particularly sockets) are available across a wider range of OS platforms compared with RPC toolkits.

The request-response and "oneway" communication mechanisms provided by RPC toolkits are not well-suited for a certain class of applications, known as "streaming" applications. Streaming applications are characterized by high-bandwidth, long-duration communication of untyped bytestreams or relatively simple datatypes that possess stringent communication performance requirements. Interactive teleconferencing, medical imaging, and video-on-demand are examples of streaming applications.

The quality of service requirements of streaming applications frequently cannot tolerate the performance overhead caused by RPC. This overhead stems from non-optimized presentation format conversions, non-optimized memory management, inefficient receiver-side demultiplexing, stop-and-wait flow control, synchronous send-side method invocations, and non-adaptive retransmission timer schemes. Traditionally, meeting the requirements of streaming applications has involved direct access to network programming interfaces such as sockets [2] or System V TLI [3].

2.3 Lower-level Programming Interfaces

Lower-level IPC mechanisms provide interfaces to an OS kernel's communication subsystem. For example, the SVR4 `putmsg` and `getmsg` system calls may be used to directly access the transport provider interface (TPI) [12] and the data-link provider interface (DLPI) [13] available in System V STREAMS [14]. It is also possible to develop network services (such as routers or network file systems) that reside entirely within the OS kernel [4]. However, programming at this level is usually not portable between different OS platforms nor often even different versions of the same OS.

2.4 Evaluation

Compared with RPC toolkits, it is more challenging to program distributed applications using transport-level or lower-level IPC interfaces. Conventional transport layer programming libraries (such as sockets and TLI) lack type-safe, portable, re-entrant, and extensible interfaces. For instance, socket endpoints are implemented via weakly-typed descriptors that increase the potential for subtle errors to occur at run-time [15]. The `IPC SAP` class category[2] provides an object-oriented interface that encapsulates much of the complexity of transport layer interfaces. The goals of `IPC SAP` are to improve the *correctness*, *ease of use*, and *portability/reusability* of communication software, without adversely affecting its performance. `IPC SAP` is currently being used on many commercial projects including Motorola, Ericsson, Bellcore, Siemens, and Credit Suisse.

3 Survey of Network Programming Interfaces

3.1 Background

This section surveys the behavior and limitations of widely available network programming interfaces. In BSD/SVR4 UNIX and Win32, the communication protocol stacks reside

[2] A class category is a collection of components that collaborate to provide a set of related interfaces and services [16].

within the protected address space of the OS kernel. Application programs running in user-space access the kernel-resident protocol stacks via interfaces such as BSD UNIX sockets, System V UNIX TLI, or Win32 named pipes. These interfaces manage local and remote endpoints of communication by allowing applications to open connections to remote hosts, negotiate and enable/disable certain options, exchange data, and close all or part of the connections when transmissions are complete.

In UNIX, sockets and TLI are loosely modeled on the standard file I/O interface, which includes the `open`, `read`, `write`, `close`, `ioctl`, `lseek`, and `select` routines [14]. However, sockets and TLI provide additional functionality that is not supported directly by the standard UNIX file I/O interfaces. This functionality stems from certain syntactic and semantic differences between file I/O and network I/O. For example, the pathnames use to identify files on a UNIX system are not globally unique across hosts in a large-scale distributed environment. Therefore, a different naming scheme (such as IP host addresses) is used to uniquely identify network applications.

The socket and TLI interfaces provide similar functionality. They support a general-purpose interface to multiple *communication domains*. A domain specifies a protocol family and an address family. Each protocol family contains a stack of protocols that implement certain types of communication in the domain. Common protocol stacks provide reliable, bi-directional, connection-oriented, message and bytestream services (based on protocols such as TCP, TP4, and SPX), as well as unreliable, connectionless datagram service (based on protocols such as UDP, CLNP, and IPX).

An address family defines an address format (*e.g.,* the address size in bytes, number and type of fields, and order of fields) and a set of kernel-resident subroutines that interpret the address format (*e.g.,* to determine which subnet an IP datagram is destined for).

Section 3.2 gives an overview on BSD sockets, Section 3.3 briefly outlines TLI, Section 3.4 covers STREAM pipes, and Section 3.5 discusses UNIX FIFOs. A complete discussion of these interfaces is beyond the scope of this paper (see [4, 2, 6, 5, 17] for additional details).

3.2 BSD Sockets

The socket interface was originally developed in BSD UNIX to provide an interface to the TCP/IP protocol suite [2]. From an application's perspective, a socket is a local endpoint of communication that is bound to an address residing on a local or a remote host. Sockets are accessed via *descriptors*. A descriptor is an unsigned integer handle that indexes into a table of descriptors. This table is maintained in the OS kernel on a per-process basis. In UNIX, socket descriptors share the same name space as other descriptors (such as file, pipe, and terminal device descriptors). Descriptors provide an encapsulation mechanism that shields applications from knowledge of internal kernel data structures. Descriptor tables reside within an OS kernel. Thus, hardware memory protection mechanisms are used to prevent accidental corruption by applications. Note that this type of encapsulation is fairly heavyweight since an application must perform a mode switch to change from user-mode to kernel-mode.

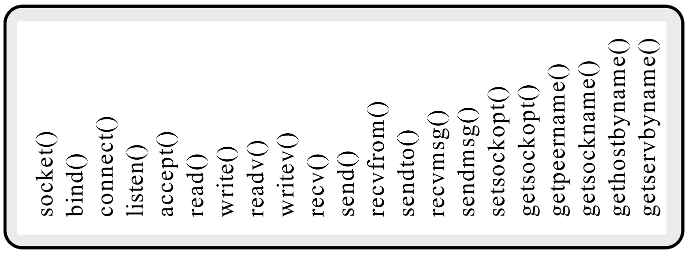

Figure 2: Socket Interface

The socket interface shown in Figure 2 contains around two dozen system calls that may be classified into the following general categories:

• **Local context management:** The socket interface provides the following routines for managing local context information:

- `socket` – which allocates the lowest unused socket descriptor;
- `bind` – which associates a socket descriptor with a local or remote address;
- `getsockname` and `getpeername` – which determine the local or remote address, respectively, a socket is connected with;
- `close` – which deallocates a socket descriptor, making it available for subsequent reuse.

• **Connection establishment and connection termination:** The socket interface provides the following routines for establishing and terminating connections:

- `connect` – a client typically uses `connect` to *actively* establish a connection with a server;
- `listen` – a server uses `listen` to indicate it willingness to listen *passively* for incoming client connection requests;
- `accept` – a server uses `accept` to create a new endpoint of communication to service a client;
- `shutdown` – which selectively terminates the read-side and/or write-side stream of a bi-directional connection.

• **Data transfer mechanisms:** The socket interface provides the following routines that send and receive data:

- `read/write` – which receive and transfer buffers of data via a particular descriptor;

- send/recv – which are similar to read/write, though they provide an extra parameter that controls certain socket-specific operations (such as exchanging "urgent" data or "peeking" at data in a receive queue without removing it from the queue);
- sendto/recvfrom – which exchange connectionless datagrams;
- readv/writev – which support scatter-read and gather-write semantics, respectively (these operations optimize user/kernel mode switching and simplify memory management);
- sendmsg/recvmsg – which are general-purpose routines that subsume the behavior of all the other data transfer routines. For UNIX domain sockets, the sendmsg and recvmsg routines also provide the ability to pass "access rights" (such as open file descriptors) between arbitrary processes on the same host machine.

Note that these interfaces may also be used for other types of I/O such as files and terminals.

• **Options management:** The socket interface defines the following routines that allow users to alter the default semantics of socket behavior:

- setsockopt and getsockopt – which modify or query options within different layers in a protocol stack. Options include multicasting, broadcasting, and setting/getting the size of send and receive transport buffers;
- fcntl and ioctl – which are UNIX system calls that enable asynchronous I/O, non-blocking I/O, and urgent message delivery on sockets.

In addition to the socket routines describe above, communication software may use the following standard library routines and system calls:

- gethostbyname and gethostbyaddr – which handle various aspects of network addressing such as mapping host names to IP addresses;
- getservbyname – which identifies services by their port numbers or humanly-readable names;
- ntohl, ntohs, htonl, htons – which perform network byte-order transformations;
- select – which performs I/O-based and timer-based event demultiplexing on sets of open descriptors.

3.3 System V TLI

TLI is an alternative interface for accessing communication protocol stacks. TLI provides basically the same set of services that sockets does. However, it places greater emphasis on shielding applications from the details of the underlying transport provider. [4] discusses TLI in detail.

3.4 STREAM Pipes

STREAM pipes are an enhancement to the original UNIX pipe mechanism. Earlier generation UNIX pipes provided a single uni-directional stream of bytes from a writer endpoint to a reader endpoint. STREAM pipes support bi-directional delivery of bytestream and prioritized message data between processes and/or threads executing on the same host machine [17]. Although the pipe system call interface remains the same, STREAM pipes offer additional functionality that is roughly equivalent to UNIX-domain SOCK_STREAM sockets. They are somewhat more flexible than UNIX-domain sockets, however, since they enable STREAM modules to be "pushed" and "popped" to and from pipe endpoints.

By default, a STREAM pipe provides only a single channel of data between its two endpoints. Therefore, if multiple senders write to the pipe all the messages are placed into the same communication channel. This is often too restrictive since it is hard to multiplex data from multiple clients over a single channel. For example, each messages must include an identifier that enables the receiver to determine which sender transmitted which message. By using mounted STREAM pipes and the connld module [18], applications may dedicate a separate non-multiplexed I/O channel between a server and each instance of a client.

STREAM pipes and connld work as follows. The server invokes the pipe system call, creating a bi-directional endpoint of communication. The fattach system call mounts a pipe descriptor at a designated location in the UNIX file system. A server application may be created by pushing the connld STREAM module onto the mounted end of the STREAM pipe. A client application running on the same host machine as the server subsequently opens the filename associated with the mounted pipe. At this point, the connld module ensures that the client and server each receive a unique I/O descriptor identifying a non-multiplexed, bi-directional channel of communication.

3.5 FIFOs

UNIX FIFOs (also called named pipes) are a restricted form of a STREAM pipe. Unlike STREAM pipes, FIFOs offer only a uni-directional data channel from one or more senders to a single receiver. Moreover, messages from different senders are all placed into the same communication channel. Therefore, some type of demultiplexing identifier must be included explicitly in each message to enable the receiver to determine which sender transmitted the message. The STREAMS-based implementation of FIFOs in SVR4 UNIX provides both message and bytestream data delivery semantics. In contrast, earlier versions of UNIX (such as SVR3 and SunOS 4.x) only provide bytestream-oriented FIFOs. Therefore, unless fixed length messages are always used, each message sent via a FIFO must be distinguished by some form of byte count or special termination symbol. This allows a receiver to extract messages from the FIFO bytestream. FIFOs are described further in [4, 5, 17].

```
#include <sys/types.h>
#include <sys/socket.h>

const int PORT_NUM = 10000;

int buggy_echo_server (void)
{
  sockaddr s_addr;
  int      length; // (1) uninitialized variable.
  char     buf[BUFSIZ];
  int      s_fd, n_fd;
  // Create a local endpoint of communication.
  if (s_fd = socket (PF_UNIX, SOCK_DGRAM, 0) == -1)
    return -1;
  // Set up the address information to become a server.
  // (2) forgot to "zero out" structure first...
  s_addr.sin_family = AF_INET;
  // (3) used the wrong address family ...
  s_addr.sin_port = PORT_NUM;
  // (4) forgot to use htons() on PORT_NUM...
  s_addr.sin_addr.s_addr = INADDR_ANY;
  if (bind (s_fd, (sockaddr *) &s_addr,
      sizeof s_addr) == -1)
    perror ("bind"), exit (1);
  // (5) forgot to call listen()

  // Create a new endpoint of communication.
  // (6) doesn't make sense to accept a SOCK_DGRAM!
  if (n_fd = accept (s_fd, &s_addr, &length) == -1) {
    // (7) Omitted a crucial set of parens...
    int n;
    // (8) doesn't make sense to read from the s_fd!
    while ((n = read (s_fd, buf, sizeof buf)) > 0)
      // (9) forgot to check for "short-writes"
      write (n_fd, buf, n);
    // Remainder omitted...
  }
}
```

Figure 3: Buggy Echo Server

3.6 Limitations with Existing IPC interfaces

Sockets, TLI, STREAM pipes, and FIFOs provide a wide range of interfaces for accessing local and remote IPC mechanisms. These interfaces share several limitations. This following discussion focuses on sockets, though many of the limitations also apply to the other network programming interfaces, as well.

• **High potential for error:** In UNIX and Win32, descriptors for sockets, files, pipes, terminals, and other devices are identified using "weakly-typed" integer or pointer values. This weak type checking enables subtle errors to occur at run-time. For example, the socket interface cannot enforce the correct use of socket routines for different communication roles (such as active vs. passive connection establishment or datagram vs. stream communication). Moreover, a compiler cannot detect or prevent erroneous use of descriptors since descriptors are weakly typed. Thus, operations may be applied incorrectly on descriptors (*e.g.,* invoking a data transfer operation on a descriptor set up for establishing connections).

Figure 3 depicts the following subtle (and all-too-common) bugs that occur when using the socket interface:

1. Forgetting to initialize the `len` parameter to `accept` to the size of `struct sockaddr_in`;
2. Forgetting to initialize all bytes in the socket address structure to "0";
3. Using an address family type that is inconsistent with the protocol family of the socket;
4. Neglecting to use the `htons` library function to convert port numbers from host byte-order to network byte-order and vice versa;
5. Omitting the `listen` system call when creating a passive-mode `SOCK_STREAM` socket;
6. Applying the `accept` function on a `SOCK_DGRAM` socket;
7. Erroneously omitting a key set of parentheses in an assignment expression;
8. Trying to read from a passive-mode socket that should only be used to accept connections;
9. Failing to properly detect and handle "short-writes" that occur due to buffering.

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

```
if (n_fd = accept (s_fd, &s_addr, &length) == -1)
```

the value of `n_fd` will always be set to either 0 or 1, depending on whether `accept () == -1`.

A deeper problem is that C data structures lack adequate abstraction. For example, the generic `sockaddr` address structure forces the use of typecasts to provide a form of inheritance for Internet domain and UNIX domain addresses. These "subclass" address structures (`sockaddr_in` and `sockaddr_un`, respectively) overlay the `sockaddr` "base class."

In general, the use of typecasts, combined with the weakly-typed descriptor-based socket interface, makes it very hard 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.

• **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 *communication roles* (such as active vs. passive connection establishment), *communication optimizations* (such as `writev` that sends multiple buffers in a single system call), and *options* for infrequently used operations such as broadcasting, multicasting, asynchronous I/O, and urgent data delivery.

Although sockets combine this functionality into a common library, the resulting interface is complex and hard to master. 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 2). This design increases the amount of effort required to learn and use sockets correctly. In particular, programmers must understand the entire the socket interface in order to use only part of it effectively.

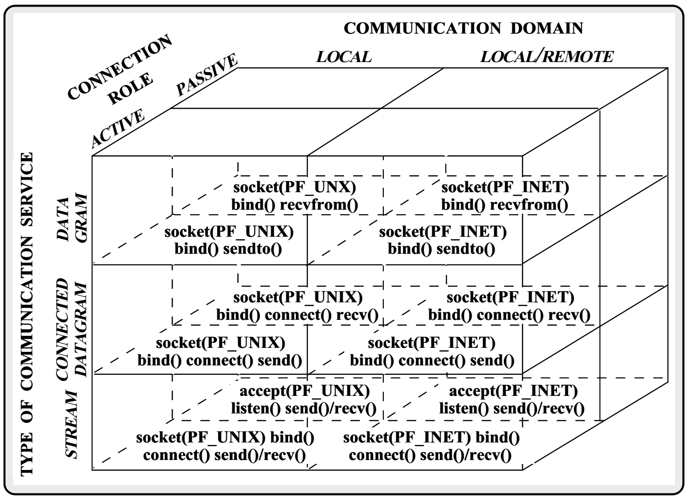

Figure 4: Socket Dimensions

If sockets are examined carefully, however, it becomes clear that the interface may be decomposed into clusters of functionality. Figure 4 clusters the related socket routines according to the following three criteria:

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

Since the interface is one-dimensional, however, this natural clustering is obscured. Section 5 illustrates how this classification is restructured into a hierarchy of classes that simplify the socket interface and enhance the type-safety of communication software.

Another problem with the socket interface is that its several dozen routines lack a uniform naming convention. Non-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` are related. Other network programming interfaces solve this problem by prepending common prefix before each routine. For example, a `t_` is prepended before each routine in the TLI library.

However, the TLI interface also contains operations with overly complex semantics. For example, unlike the socket interface, the TLI option handling interface is not specified in a standard manner. This makes it hard to write portable applications that access standard TCP/IP options. Likewise, subtle application-level code is necessary to handle the non-intuitive, error-prone behavior of `t_listen` and `t_accept` in a concurrent server with a `qlen` > 1 [4].

4 Interprocess Communication Service Access Points (IPC SAP)

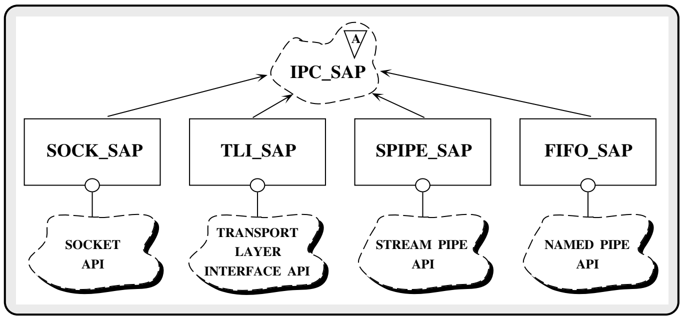

Figure 5: IPC SAP Class Category Relationships

4.1 Overview

`IPC SAP` encapsulates the standard descriptor-based IPC interfaces such as sockets, TLI, STREAM pipes, and FIFOs. As shown in Figure 5, `IPC SAP` is designed as a forest of class categories that includes `SOCK SAP` (which encapsulates the socket interface), `TLI SAP` (which encapsulates the TLI interface), `SPIPE SAP` (which encapsulates the UNIX SVR4 STREAM pipe interface), and `FIFO SAP` (which encapsulates the UNIX FIFO interface).

Each class category is organized as an inheritance hierarchy. Every subclass provides a well-defined interface to a subset of existing IPC mechanisms. Together, the subclasses within a hierarchy comprise the overall functionality of a particular communication abstraction (such as the Internet-domain or UNIX-domain protocol families). This section describes the design goals of `IPC SAP`, outlines its class categories, and discusses the principles that underly its object-oriented design.

4.2 IPC SAP Design Goals

`IPC SAP` is designed to improve the *correctness*, *ease of learning* and *ease of use*, *portability*, and *reusability* of communication software, while maintaining a high level of performance. This section discusses how `IPC SAP` achieves these goals.

• **Improve correctness:** Several of the problems with sockets are related to its weak typechecking. `IPC SAP` improves the correctness of application networking code by permitting only "type-safe" operations on instances of its classes. To enforce type-safety, `IPC SAP` ensures all of its objects are properly initialized via constructors. In addition, only well-defined operations are permitted on `IPC SAP` objects. `IPC SAP` is specifically designed to prevent accidental violations of type-safety. For example, components in the `SOCK SAP` class category prevent the accidental use of connection-oriented operations on datagram objects. Therefore, it is impossible to invoke the `accept` method on a datagram object, `recv` or `send` data on a connector or acceptor factory object, or invoke a `sendto` operation on a connection-oriented object.

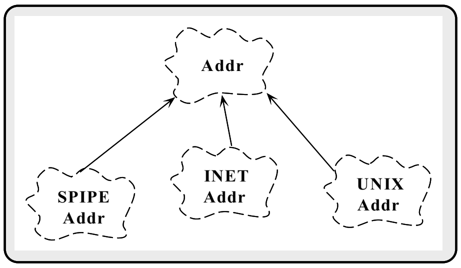

Figure 6: The IPC SAP Address Class Hierarchy

Since IPC SAP classes are strongly typed, any attempts to perform invalid operations are rejected at compile-time rather than at run-time. This point is illustrated in the revised SOCK SAP revision of buggy_echo_server shown in Figure 14. This example fixes all the problems with sockets identified in Figure 3.

• **Enhance ease of learning and ease of use:** Simplifying the use of common IPC operations is a goal related to correctness. By providing simpler interfaces, developers are able to concentrate on writing applications, instead of wrestling with the networking code. In general, IPC SAP simplifies its network programming interfaces as follows:

- *Designing auxiliary classes that shield applications from error-prone details – e.g.,* IPC SAP contains the Addr class hierarchy (shown in Figure 6).[3] This hierarchy supports several diverse network addressing formats via a type-secure C++ interface. The Addr hierarchy eliminates several common programming errors (such as forgetting to zero-out a sockaddr addressing structure) associated with using the C-based struct sockaddr data structures directly;
- *Combining several operations to form a single operation – e.g.,* the ACE_SOCK_Acceptor is a factory for passive connection establishment. Its constructor performs the various socket system calls (such as socket, bind, and listen) required to create a passive-mode server endpoint;
- *Supplying default parameters for typical method argument values – e.g.,* the addressing parameters to accept are frequently NULL pointers. To simplify programming, these values are given as defaults in SOCK_Acceptor::accept, so that programmers don't have to provide them.
- *Utilizing traits to convey "metaclass" information – e.g.,* every IPC SAP class contains a uniform set of traits. These traits are typedef'd to designate the address class (*e.g.,* ACE_INET_Addr) and/or stream class (*e.g.,* ACE_TLI_Stream) associated with each IPC SAP type, as follows:

```
class ACE_SOCK_Connector : public ACE_SOCK
{
public:
  // Traits
  typedef ACE_INET_Addr PEER_ADDR;
  typedef ACE_SOCK_Stream PEER_STREAM;
  // ...
};

class ACE_TLI_Connector : public ACE_SOCK
{
public:
  // Traits
  typedef ACE_INET_Addr PEER_ADDR;
  typedef ACE_TLI_Stream PEER_STREAM;
  //...
};
```

As shown in Section 6, the use of traits in conjunction with C++ parameterized types supports a powerful design paradigm known as "generic programming" [19].

[3] To reduce clutter, the ACE_ prefix has been omitted from all ACE IPC SAP class names shown in this and other figures.

• **Increase reusability:** Inheritance-based hierarchical decomposition is used in IPC SAP to increase the amount of common code that is shared amongst the various IPC mechanisms (such as the C++ interface to the lower-level OS device control system calls like fcntl and ioctl). Inheritance enhances reuse within the IPC SAP implementation by sharing code between different subclasses. For example, the IPC SAP root base class provides standard methods and data that is shared by the other derived classes. These shared components provide an descriptor variable and its related set/get methods, as well as methods that enable and disable asynchronous I/O, non-blocking I/O, and urgent message delivery on the descriptor.

• **Portability:** Several C++ features are used to enhance portability. For example, IPC SAP provides a platform-independent interface that improves communication software portability by using C++ templates. As illustrated in Figure 7, a subset of the SOCK SAP and TLI SAP classes offer the same object-oriented interface. Each platform may possess a different underlying interface for local and remote network programming (*e.g.,* sockets vs. TLI). However, it is possible to write applications that may be transparently parameterized with either class. This enhances application portability across OS platforms that may not support both sockets and TLI.

The use of classes (as opposed to stand-alone functions) helps to simplify network programming by allowing applications to be parameterized by the type of IPC mechanism they requires. As discussed in Section 7, parameterization helps to improve portability among platforms that support different network programming interfaces (such as sockets or TLI).

• **Performance:** To encourage developers to substitute IPC SAP for existing interfaces, the implementation is designed to operate efficiently. Several techniques help improve performance without sacrificing clarity and modularity. First, most methods are specified as C++ "inline functions." This eliminates the extra run-time overhead of calling IPC SAP methods. Inlining is a reasonable approach since each method is very short (averaging approximately 3 lines per method). Second, virtual functions are not used in the IPC SAP inheritance hierarchy. This improves performance since

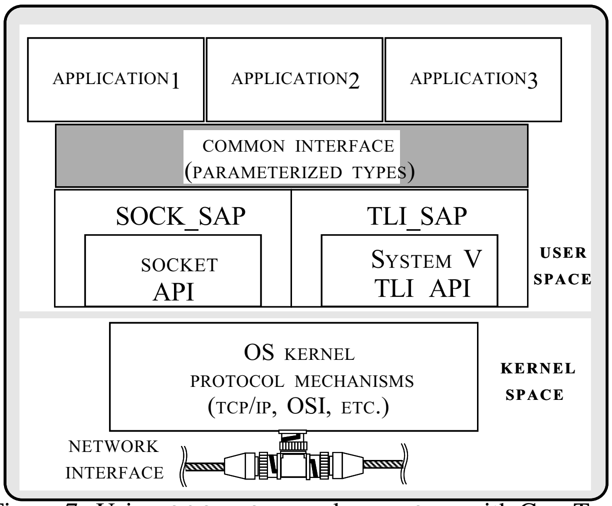

Figure 7: Using `SOCK SAP` and `TLI SAP` with C++ Templates to Increase Portability

(1) it eliminates indirect `vtable` function pointer dispatching and (2) it facilitates the direct inlining of certain short, frequently-accessed methods (such as sending and receiving user data).

5 The Object-Oriented Design of IPC SAP

This section describes the object-oriented design of the C++ class categories that comprise the IPC SAP class categories. Particular emphasis is given to the design of the `SOCK SAP` C++ wrapper for sockets. `SOCK SAP` has been ported to many UNIX platforms, as well as to the WinSock transport layer programming interface available on Windows, Windows NT, Chicago, and OS/2 [20].

5.1 SOCK SAP

The `SOCK SAP` class category provides applications with an object-oriented interface to the Internet-domain and UNIX-domain protocol families [5]. Applications may access the functionality of the underlying Internet-domain or UNIX-domain socket types by inheriting or instantiating the appropriate `SOCK SAP` subclasses shown in Figure 8. The `ACE_SOCK*` subclasses encapsulate Internet-domain functionality and the `ACE_LSOCK*` subclasses encapsulate UNIX-domain functionality. As shown in Figure 8, the subclasses may be further decomposed into the following components:

- `*Dgram` vs. `*Stream` subclasses – the `*Dgram` subclasses provide unreliable, connectionless, message functionality whereas the `*Stream` components provide reliable, connection-oriented, bytestream functionality;
- `*Acceptor` and `*Connector` vs. `*Stream` subclasses – passive and active connection establishment functionality is provided to servers by the `*Acceptor` and `*Connector` subclasses, respectively whereas the `*Stream` components provide bidirectional bytestream data transfer functionality used by applications that play both client and server roles.

Using C++ wrappers to encapsulate the socket interface helps to (1) detect many subtle application type system violations at compile-time, (2) facilitate a platform-independent transport-level interface that improves application portability, and (3) greatly reduce the amount of application code and development effort expended upon lower-level network programming details. To illustrate the latter point, the following example program implements a simple client application. This application uses the `ACE_SOCK_Dgram_Bcast` class to broadcast a message to all servers listening on a designated port number in a LAN subnet:

```
int
main (int argc, char *argv[])
{
  ACE_SOCK_Dgram_Bcast b_sap (ACE_Addr::sap_any);
  char *msg;
  u_short b_port;

  msg = argc > 1 ? argv[1] : "hello world\n";
  b_port = argc > 2 ? atoi (argv[2]) : 12345;

  if (b_sap.send (msg, strlen (msg),
                  b_port) == -1)
    perror ("can't send broadcast");
  return 0;
}
```

It is instructive to compare this concise example with the dozens of lines of C source code required to implement broadcasting using the socket interface directly.

5.2 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. Moreover, `SOCK SAP` enhances these qualities without sacrificing performance [1]. This section outlines the software architecture of `SOCK SAP` and explains the classes used by the programming examples in Section 6. Readers who are not interested in this level of detail may want to skip to Section 7, 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 [16] in Figure 8. Dashed clouds indicate classes and directed edges indicate inheritance relationships between these classes (*e.g.,* `ACE_SOCK_Stream` inherits from `ACE_SOCK`). The general structure of `SOCK SAP` corresponds to the taxonomy of *communication services*, *connection roles*, and *communication*

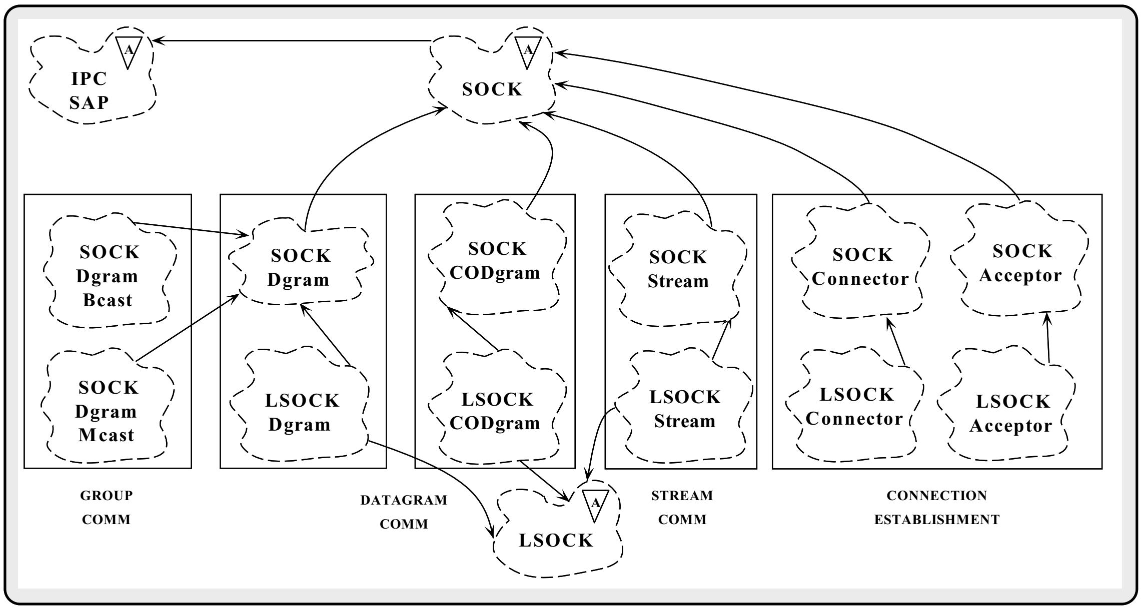

Figure 8: The SOCK SAP Class Categories

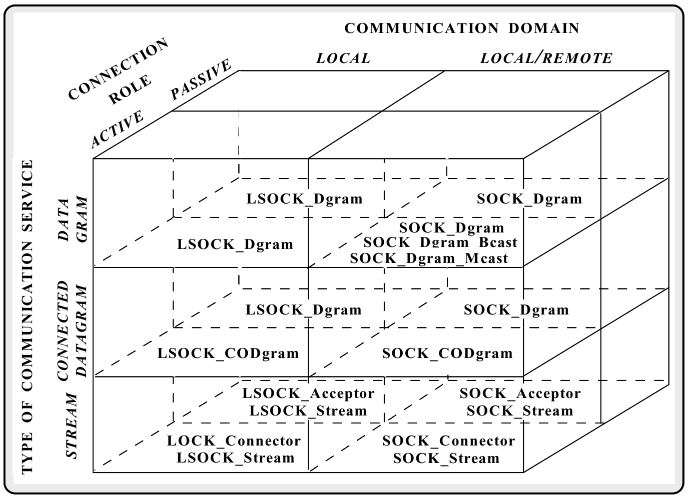

Figure 9: Taxonomy of SOCK SAP Classes and Communication Dimensions

domains shown in Figure 9. It is instructive to compare Figure 4 with Figure 9. 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 8.

5.2.1 Base Classes

The `IPC SAP`, `ACE_SOCK`, and `ACE_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 `ACE_LSOCK` to obtain this functionality.

`SOCK SAP` distinguishes between the `ACE_LSOCK*` and `ACE_SOCK*` classes on the basis of network address formats and communication semantics. In particular, the `ACE_LSOCK*` classes use UNIX pathnames as addresses and allow only intra-machine IPC. The `ACE_SOCK*` classes, on the other hand, use Internet Protocol (IP) addresses and port numbers and allow both intra- and inter-machine IPC.

5.2.2 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* [21]. The structure of passive/active connection establishment and data transfer relationships are captured by the following connection-oriented `SOCK SAP` classes:

- **ACE_SOCK_Acceptor and ACE_LSOCK_Acceptor:** The `*Acceptor` classes are factories [22] that passively establish new endpoints of communication in response to active connection requests. The `ACE_SOCK_Acceptor` and `LSOCK_Acceptor` produce `ACE_SOCK_Stream` and `ACE_LSOCK_Stream` connection endpoint objects, respectively.

- **ACE_SOCK_Connector and ACE_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 `ACE_SOCK_Connector` and `ACE_LSOCK_Connector` factories produce `ACE_SOCK_Stream` and `ACE_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.

5.2.3 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:

- **ACE_SOCK_Stream and ACE_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; `ACE_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` or `recv` may write or read less, respectively, 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.

5.2.4 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 [5]) 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:

- **ACE_SOCK_Dgram and ACE_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. `ACE_LSOCK_Dgram` inherits all the operations of both `ACE_SOCK_Dgram` and `ACE_LSOCK`. It only exchanges datagrams between processes on the same host. The `ACE_SOCK_Dgram` class, on the other hand, may exchange datagrams between processes on local and/or remote hosts.

- **ACE_SOCK_CODgram and ACE_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). `ACE_SOCK_CODgram` inherits mechanisms from the `ACE_SOCK` base class. `ACE_LSOCK_CODgram` inherits mechanisms from both `ACE_SOCK_CODgram` and `ACE_LSOCK` (which provides the ability to pass file handles).

5.2.5 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:

- **ACE_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.

- **ACE_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.

5.3 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 6 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 `ACE_LSOCK*` classes, the `ACE_INET_Addr` subclass is associated with the `ACE_SOCK*` and `ACE_TLI*` classes, and the `SPIPE Addr` subclass is associated with the STREAM pipe wrappers in `SPIPE SAP`.

5.4 TLI SAP

The `TLI SAP` class category provides a C++ interface to the System V Transport Layer Interface (TLI). The `TLI SAP` inheritance hierarchy for TLI is almost identical to the `SOCK SAP` C++ wrapper for sockets. The primary difference is that TLI and `TLI SAP` do not define an interface to the UNIX-domain protocol family. By combining C++ features (such as default parameter values and templates) together with the `tirdwr` (the `read/write` compatibility STREAMS module), it becomes relatively straight-forward to develop applications that may be parameterized at compile-time to operate correctly over either a socket-based or TLI-based transport interface.

The following code illustrates how C++ templates may be applied to parameterize the IPC mechanisms used by an application. This code was extracted from the distributed logging facility described in [23]. In the code below, a subclass derived from `Event Handler` is parameterized by a particular type of transport interface and its corresponding protocol address class:

```
// Logging_Handler header file.
template <class PEER_STREAM>
class Logging_Handler : public Event_Handler
{
public:
  Logging_Handler (void);
  virtual ~Logging_Handler (void);

  virtual int handle_close (int);
  virtual int handle_input (int);
  virtual int get_handle (void) const
  {
    return this->xport_sap.get_handle ();
  }

protected:
  PEER_STREAM xport_sap;
};
```

Depending on certain properties of the underlying OS platform (such as whether it is BSD-based SunOS 4.x or System V-based SunOS 5.x), the logging application may instantiate the `Client Handler` class to use either `SOCK SAP` or `TLI SAP`, as shown below:

```
#if defined (MT_SAFE_SOCKETS)
typedef ACE_SOCK_Stream PEER_STREAM;
#else
typedef ACE_TLI_Stream PEER_STREAM;
// Logging application.
#endif // MT_SAFE_SOCKETS.

class Logging_Handler :
  public Logging_Handler<PEER_STREAM>
{
  // ...
};
```

The increased flexibility offered by this template-based approach is extremely useful when developing an application that must run portability across multiple OS platforms. In particular, the ability to parameterize applications according to transport interface is necessary across variants of SunOS platforms since the socket implementation in SunOS 5.2 is not thread-safe and the TLI implementation in SunOS 4.x contains a number of serious defects.

TLI SAP also shields applications from many peculiarities of the TLI interface. For example, the accept method in the ACE_TLI_Acceptor class encapsulates the subtle application-level code required to handle the non-intuitive, error-prone behavior of t_listen and t_accept in a concurrent server with a qlen > 1 [4]. The accept method passively establishes client connection requests. Through the use of C++ default parameter values, the standard method for calling the accept method is syntactically equivalent for both TLI SAP-based and SOCK SAP-based applications.

5.5 SPIPE SAP and FIFO SAP

The SPIPE SAP class category provides a C++ wrapper interface for mounted STREAM pipes and connld [18]. The SPIPE SAP inheritance hierarchy mirrors the one used for SOCK SAP and TLI SAP. It offers functionality that is similar to the SOCK SAP ACE_LSOCK* classes (which themselves encapsulate UNIX-domain sockets). However, SPIPE SAP is more flexible than the ACE_LSOCK* interface since it enables STREAM modules to be "pushed" and "popped" to and from SPIPE SAP endpoints, respectively. SPIPE SAP also supports bi-directional delivery of bytestream and prioritized message data between processes and/or threads executing within the same host machine [17].

The FIFO SAP class category encapsulates the UNIX FIFO mechanism.

6 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 [1]. For comparison, this application is also written with sockets. 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 10 and 11 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 11 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 10 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 10 and 11 is not portable to platforms that do not support both sockets and select.

Figures 12 and 13 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 10 and Figure 11. 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 6, which hides the subtle and error-prone details that must be programmed explicitly in Figures 10 and 11. Moreover, the low-level details of non-blocking connection establishment are performed by the SOCK Connector factory. In addition, the use of template *traits* minimizes the number of type parameters that must be specified when instantiated the parameterized functions.
- *Increased typesafety – e.g.,* the ACE_SOCK_Acceptor and ACE_SOCK_Connector connection factories create ACE_SOCK_Stream objects. This prevents the type errors shown in Figure 3 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 ACE_SOCK_Acceptor and ACE_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.,* due to the use of template traits, switching between sockets and TLI simply requires changing

  ```
  send_data <ACE_TLI_Connector> (s_addr);
  ```

  in the client to

  ```
  send_data <ACE_SOCK_Connector> (s_addr);
  ```

  and

  ```
  recv_data<ACE_SOCK_Acceptor> (s_addr);
  ```

  in the server to

  ```
  recv_data<ACE_TLI_Acceptor> (s_addr);
  ```

  As shown in Section 7, conditional compilation directives can be used to further decouple the communication software from reliance upon a particular type of network programming interface.

7 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

```
#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 10: 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 11: Socket-based Server Example

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

// SOCK_SAP Client.

template <class CONNECTOR>
void send_data (CONNECTOR::PEER_ADDR peer_addr)
{
  // Data transfer object.
  CONNECTOR::PEER_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.
  ACE_INET_Addr s_addr (port_num, host)

  // Use SOCK SAP wrappers on client's side.
  send_data <ACE_SOCK_Connector> (s_addr);
  return 0;
}
```

Figure 12: SOCK SAP-based Client Example

```
static const int PORT_NUM = 10000;

// SOCK_SAP Server.

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

  // Data transfer object.
  ACCEPTOR::PEER_STREAM peer_stream;

  // Remote peer address.
  ACCEPTOR::PEER_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.
  ACE_INET_Addr s_addr (port_num);

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

Figure 13: SOCK SAP-based Server Example

- 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 3.6 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 14. This version fixes the problems with sockets and C identified in Figure 3. 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 `ACE_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.

• **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 `ACE_SOCK_Connector` constructor has six parameters:

```
ACE_SOCK_Connector
  (ACE_SOCK_Stream &new_stream,
   const ACE_SOCK_Addr &remote_sap,
   ACE_Time_Value *timeout = 0,
   const ACE_SOCK_Addr &local_sap =
     (ACE_SOCK_Addr &) Addr::sap_any,
   int protocol_family = PF_INET,
   int protocol = 0);
```

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

```
ACE_SOCK_Stream stream;

// Compiler supplies default values.
ACE_SOCK_Connector con (stream,
                        ACE_INET_Addr (port, host));
// ...
```

Therefore, to simplify programming, the values are given as defaults in the `ACE_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 `ACE_SOCK_Acceptor` class only permits operations that apply for programs playing passive roles and the `ACE_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 `ACE_SOCK_SAP` compared with using the highly-general UNIX `sendmsg/recvmsg` routines. For example, using `ACE_LSOCK*` classes to pass socket descriptors is very concise:

```
ACE_LSOCK_Stream stream;
ACE_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;
```

```
int echo_server (ACE_INET_Addr s_addr)
{
  // Initialize the passive mode server.
  ACE_SOCK_Acceptor acceptor (s_addr);

  // Data transfer object.
  ACE_SOCK_Stream peer_stream;

  // Client remote address object.
  ACE_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 14: `SOCK SAP` Revision of the Echo Server

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

In contrast, the `ACE_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:

```
ACE_SOCK_Acceptor acceptor ((ACE_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 8). 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 `ACE_LSOCK` class (described in Section 5.2) from the inheritance hierarchy without affecting the interfaces of other classes in the `SOCK SAP` design.

```
template <class ACCEPTOR>
int echo_server (ACCEPTOR::PEER_ADDR s_addr)
{
  // Initialize the passive mode server.
  ACCEPTOR acceptor (s_addr);

  // Data transfer object.
  ACCEPTOR::PEER_STREAM peer_stream;

  // Remote address object.
  ACCEPTOR::PEER_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 15: Template Version of the Echo Server

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 15 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 ACE_SOCK_Acceptor ACCEPTOR;
#else // USE_TLI
typedef ACE_TLI_Acceptor ACCEPTOR;
#endif // USE_SOCKETS.

const int PORT_NUM = 10000;
```

```
int main (void)
{
  // ...

  // Invoke the echo_server with appropriate
  // network programming interfaces (note the
  // use of template traits).
  echo_server<ACCEPTOR> (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).

For example, the `SOCK SAP` and `TLI SAP` classes offer the same object-oriented interface (depicted in Figure 7). Certain OS platforms may possess different underlying network programming interfaces such as sockets but not TLI or vice versa. Using `IPC SAP`, applications can be written that are transparently parameterized with either the `SOCK SAP` or `TLI SAP` class category. C++ templates support a loose form of type conformance that does not constrain an interface to encompasses *all* potential functionality. Instead, templates are used to parameterize application code that is carefully designed to invoke only a subset of methods that are common to the various communication abstractions (*e.g.,* `open`, `close`, `send`, `recv`, etc.).

The type abstraction provided by templates improves portability among platforms that support different network programming interfaces (such as sockets or TLI). For example, the parameterizing the transport interface turned out to be useful for developing applications across various SunOS platforms. The socket implementation in SunOS 5.2 was not thread-safe and the TLI implementation in SunOS 4.x contains a number of serious defects.

• **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 `ACE_SOCK_Stream` `recv` and `send` methods) are specified as C++ inline functions to eliminate run-time function call 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 6). 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 `ACE_INET_Addr` automatically zeros-out the `sockaddr` addressing structure and converts the port number to network byte order, as follows:

```
class ACE_INET_Addr : public ACE_Addr
{
public:
  ACE_INET_Addr::ACE_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_;
};
```

8 Summary

`IPC SAP` provides a family of object-oriented C++ wrappers that encapsulate standard local and remote IPC mechanisms available on contemporary operating systems. These encapsulated interfaces simplify the development of communication software by making it easier to write correct, compact, portable, and efficient code. In addition, the wrapper methodology facilitates organizational transition to C++ by (1) incrementally teaching developers object-oriented design principles and (2) leveraging off of an existing code base in languages other than C++. This paper concludes by describing several advantages and disadvantages of using C++ to implement `IPC SAP` and outlining future papers that explore additional uses of `IPC SAP`.

Advantages and Disadvantages of Using C++: The primary advantages of C++ to develop wrappers include the following:

- Classes hide differences in addressing formats (*e.g.,* Internet vs. UNIX-domain addressing) and encapsulate distinct interface behaviors in different classes (*e.g.,* an `ACE_SOCK_Acceptor` object's interface is specially-tailored for server operations);
- Inheritance support subsets of functionality (*e.g.,* omitting the ability to pass file descriptors on systems that do not support this feature);
- Templates enable the parameterization of different IPC mechanisms into applications, which improves portability across platforms.

One disadvantage of C++ is its lack of a widely-available exception handling mechanism. The lack of exception handling complicates error recovery and reduces type-safety. For example, if a constructor fails (*e.g.,* if a user-specified remote address does not correspond to an actual host) it is possible to begin using an `IPC SAP` object without initializing it properly. Hopefully, this problem will be solved over time as the ANSI/ISO C++ exception handling mechanism becomes available on most OS platforms.

Current Status and Future Topics: `IPC SAP` is available with the ADAPTIVE Communication Environment (ACE) [24]. ACE is a collection of reusable C++ class libraries and object-oriented framework components that enhance the development of communication software. All the source code described in this paper is available online at `http://www.cs.wustl.edu/~schmidt/ACE.html`.

ACE has been ported to many versions of UNIX and Windows NT and is currently being used in many commercial products including the Bellcore 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 and Siemens Medical Engineering.

Subsequent papers in this series will illustrate other uses of the `IPC SAP` class categories. For example, the following paper describes a related communication software development tool called the `Reactor`. The `Reactor` encapsulates the functionality of the `select` and `poll` event demultiplexing mechanisms within a portable and extensible C++ wrapper [25, 26]. `Select` and `poll` are UNIX system calls that detect the occurrence of different types of input and output events on one or more I/O descriptors simultaneously. Together, `IPC SAP` and the `Reactor` provide an efficient and extensible suite of reusable software components that simplify the development of correct, concise, portable, and efficient communication software.

References

[1] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the* 1^{st} *Conference on Object-Oriented Technologies and Systems*, (Monterey, CA), USENIX, June 1995.

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

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

[4] S. Rago, *UNIX System V Network Programming*. Reading, MA: Addison-Wesley, 1993.

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

[6] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

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

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

[9] W. Rosenberry, D. Kenney, and G. Fischer, *Understanding DCE*. O'Reilly and Associates, Inc., 1992.

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

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

[12] OSI Special Interest Group, *Transport Provider Interface Specification*, December 1992.

[13] OSI Special Interest Group, *Data Link Provider Interface Specification*, August 1991.

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

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

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

[17] W. R. Stevens, *Advanced Programming in the UNIX Environment*. Reading, Massachusetts: Addison Wesley, 1992.

[18] D. L. Presotto and D. M. Ritchie, "Interprocess Communication in the Ninth Edition UNIX System," *UNIX Research System Papers, Tenth Edition*, vol. 2, no. 8, pp. 523–530, 1990.

[19] A. Stepanov and M. Lee, "The Standard Template Library," Tech. Rep. HPL-94-34, Hewlett-Packard Laboratories, April 1994.

[20] *Windows Sockets - An Open Interface for Network Programming under Microsoft Windows*, Version 1.1 ed., January 1993.

[21] 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.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[23] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[24] 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.

[25] D. C. Schmidt, "The Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing (Part 1 of 2)," *C++ Report*, vol. 5, February 1993.

[26] D. C. Schmidt, "The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2)," *C++ Report*, vol. 5, September 1993.