Systems Programming with C++ Wrappers
Encapsulating Interprocess Communication Mechanisms with Object-Oriented Interfaces

Douglas C. Schmidt
schmidt@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine, CA 92717, (714) 856-4105

An earlier version of this paper appeared in the September/October 1992 issue of the C++ Report.

1 Introduction

This article is the first in a series that describes techniques for encapsulating operating system (OS) interprocess communication (IPC) mechanisms within object-oriented (OO) C++ wrappers. These OS mechanisms include support for both local-host IPC (such as semaphores; message queues; shared memory; memory-mapped files; named, unnamed, and STREAM pipes; and BSD UNIX-domain sockets) and network IPC (such as remote procedure calls (RPC); BSD Internet-domain sockets; and the System V Transport Layer Interface (TLI)). The primary motivations for using C++ wrappers are (1) to simplify the development of correct, concise, portable, and efficient applications and (2) to facilitate the introduction of object-oriented design (OOD) and C++ into software development organizations. In addition to describing C++ wrappers, this series of articles also clarifies the semantics of several advanced UNIX and Windows NT IPC mechanisms.

This introductory article motivates the concept of C++ wrappers, describes the limitations of existing OS interfaces that wrappers help to overcome, outlines the topics that will appear in subsequent articles, defines relevant networking and IPC terms (see Table 1), and summarizes the advantages and disadvantages of using C++ as a systems programming language for applications that utilize IPC mechanisms. Subsequent articles examine several IPC mechanisms that benefit from object-oriented encapsulation. For instance, the next article in this series describes the object-oriented design and implementation of a network IPC interface known as `IPC_SAP`, which encapsulates the local and remote IPC mechanisms available in the BSD socket and System V TLI APIs. `IPC_SAP` is currently being used in the ADAPTIVE system, which is a flexible development and evaluation environment for producing customized lightweight transport-layer communication protocols [1].

2 Systems Programming and IPC

Systems software programs (such as databases, windowing systems, network file servers, compilers, linkers, editors, and device drivers) typically access and manipulate operating system resources such as I/O controllers and information located in data structures residing within an OS kernel. As distributed computing become more prevalent, an increasingly important class of OS system mechanisms involve interprocess communication (IPC) within a single host, as well as across networks and internetworks. IPC mechanisms exchange different types of bytestream-oriented and message-oriented data between processes that are executing in different address spaces on the same and/or different host machines. For example, Figure 1 illustrates a distributed application that utilizes both local and remote IPC in a multi-level client/server manner. In this scenario, application processes (*e.g.,* $P_1, P_2, and P_3$) running on the client hosts A and B send discrete messages to a local client daemon via named pipes. In turn, each client daemon forwards these messages to the remote daemon on a designated server host across the network via TCP stream connections. The server receives the messages and displays them on one or more output devices (such as a printer, persistent storage device, or monitoring console). Subsequent articles will address this client/server architecture in greater detail using the `Reactor` I/O-based and timer-based service multiplexing facility [2, 3].

2.1 Limitations with Existing Operating System Interfaces

Developing communication system software is difficult since it requires detailed knowledge of many concepts such as (1) network addressing and remote service identification, (2) creation, synchronization, and communication mechanisms for processes and threads, (3) system call *Application Programmatic Interfaces* (API)s for local and remote IPC, and (4) presentation layer conversion techniques. Moreover, applications are often required to be efficient, functional, correct, and portable across heterogeneous operating environments. In addition, popular operating systems like UNIX, VMS, and MVS were developed well before the advent of object-oriented design and programming. This becomes evident

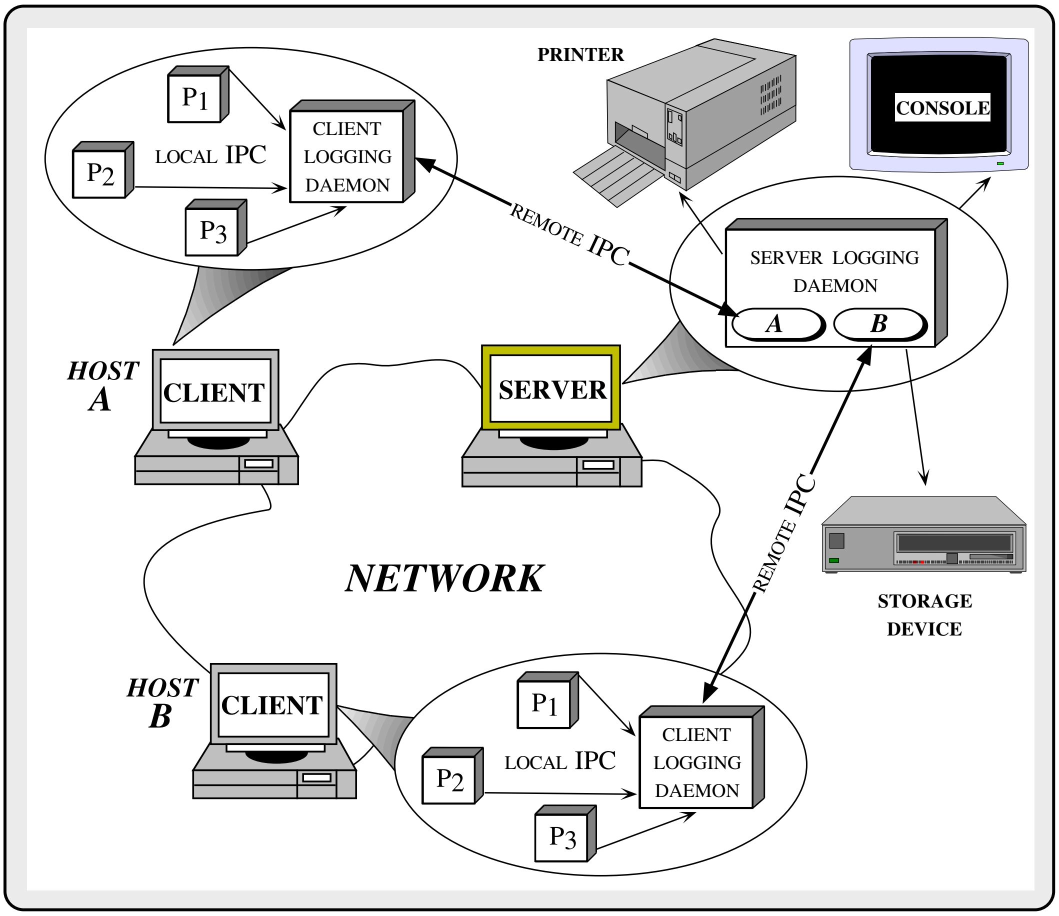

Figure 1: A Local and Remote Interprocess Communication Scenario

after examining the format of system calls and standard library routines that comprise existing OS APIs for local and remote IPC such as BSD sockets, System V Transport Layer Interface (TLI), STREAM pipes and FIFOs. In general, these APIs share several common problems:

• **Lack of Type-Security:** the UNIX and Windows NT system call API identifies particular instances of I/O devices (such as files, sockets, and terminals) using a common namespace consisting of unsigned integer I/O descriptors. This common descriptor name space is often considered a UNIX "feature" since it increases application flexibility and interoperability. For example, application code that performs a `read()` may be written identically regardless of whether the underlying device is a disk file or a network connection. However, this degree of flexibility also allows subtle errors to occur at run-time. For instance, I/O descriptors are "weakly-typed" in the sense that disk file descriptors are not syntactically distinguished from network connection descriptors. Therefore, it is easy to use the wrong descriptors in the wrong circumstances by accident. Moreover, while I/O operations on devices may look superficially equivalent, there may be additional semantics that have subtle differences (such as the occurrence of "short reads" from network connections due to flow control and/or OS buffering). Accidental misuse of descriptors would be detected at compile-time if the UNIX network IPC API enforced stronger type-checking.

• **Steep Learning Curve:** Many operating systems support multiple protocol suites (such as TCP/IP and OSI protocols). Moreover, the application-level APIs for these IPC mechanisms possess a general-purpose "one-size-fits-all" design that uses the same interface for each protocol suite. This results in a complex API that requires significant effort to learn and use effectively. One consequence of this complexity is that there is no clear association between certain related system calls. For example, the BSD socket interface consists of numerous system calls (*e.g.,* `socket()`, `listen()`, `accept()`, `connect()`, etc.). Since these system calls do not follow any standard naming convention, however, it is not immediately obvious that they are members of the same related abstraction. Moreover, conventional OS APIs have "linear," single-level interfaces. This lack of hierarchical structuring makes it difficult to determine which system calls are naturally grouped together (such as which socket calls are intended for *client-side* operations and which are used for *server-side* operations).

Figure 2: Relationship of C++ Wrappers to Other Operating System Components

• **Non-Portability:** It is often difficult to write portable code that uses OS IPC mechanisms since there are several competing "standards" to choose from, (*i.e.,* BSD sockets and System V UNIX TLI). This increases the complexity of developing and maintaining application source code. For example, achieving portability may require the use of conditional compilation that is parameterized by the host OS type.

On UNIX, the BSD socket and System V TLI APIs are semantically similar since they both offer connection-oriented and connectionless interfaces to the same protocol suites. However, they are lexically and syntactically incompatible since they possess different system call names that use different calling interfaces. This often forces developers to either (1) choose between the two APIs (which decreases application portability between different variants of BSD and System V UNIX) or (2) use conditional compilation that is parameterized by the host OS type (which increases the complexity of developing and maintaining application source code). This problem is solved to some extent in System V Release 4 since it supports both the System V and BSD APIs. However, there are many operating system vendors and OS platforms that have not upgraded.

• **Non-Extensibility:** Another limitation with existing OS APIs is that it is difficult to specialize or generalize their functionality without writing new code and/or modifying existing application code. On the other hand, APIs that use object-oriented features such as inheritance and dynamic binding are typically easy to extended transparently [3]. Extensibility becomes particularly important when dealing with C++ wrapper that provide more complex semantics (such as event multiplexing and multi-processing).

2.2 The C++ Wrapper Alternative

Switching to a consistently designed, strongly-typed, object-oriented operating system (such as the Choices OS from University of Illinois [4]) helps ameliorate some of the problems described in the preceding section. However, adopting a completely different OS is often impractical, due to factors such as the lack of application portability, platform availability, and user and developer familiarity. A more realistic alternative is to develop object-oriented interfaces that encapsulate existing OS mechanisms.

Due to the efficiency and availability of C++, it makes sense to encapsulate these existing interfaces within C++ classes and inheritance hierarchies. Developing distributed applications based upon C++ wrappers helps improve software quality factors such as correctness, ease of learning and ease of use, portability, and extensibility. In addition, it also serves as an effective method for introducing C++ and object-oriented design (OOD) into software development organizations.

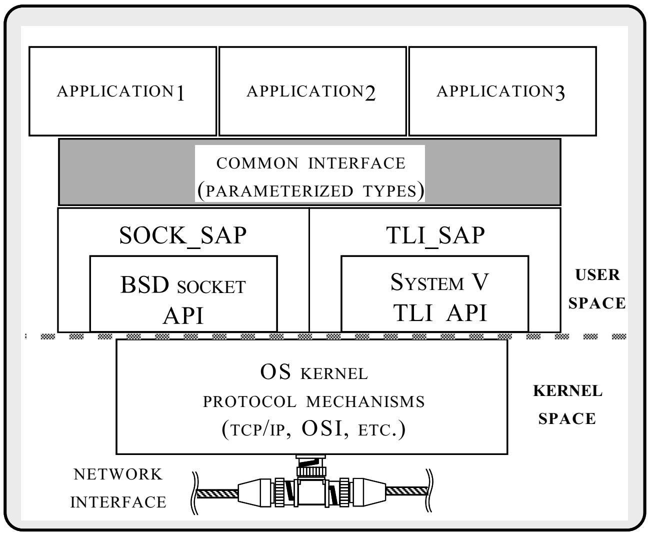

Figure 3: Using C++ Wrappers to Increase Portability

Figure 2 illustrates how C++ wrappers interact with (1) user applications and (2) the OS system call interfaces and kernel mechanisms (such as local and remote IPC, various types of file systems, and other I/O devices, etc.). In this approach, applications use C++ wrappers to access these mechanisms via type-secure, object-oriented class interfaces, rather than through the weakly-typed C language API. As shown in the figure, it is still possible to access the underlying API directly if necessary (*e.g.,* when interfacing application code with existing "non-wrapped" C library routines and system calls).

In general, C++ wrappers are designed to reduce the complexity of systems software development without compromising application performance. For example, the wrappers discussed in this series are implemented using C++ language features (such as inline functions) that minimize or even eliminate the performance overhead incurred by this additional "object-oriented layer of abstraction." In particular, inline functions avoid making an extra function call for each access to an underlying OS system call or library routine.

2.3 Benefits of C++ Wrappers

My Ph.D. research project focuses on a framework for developing and experimenting with alternative process architectures for shared memory, symmetric multi-processor platforms [5]. One portion of this project is the ADAPTIVE Communication Environment (ACE) [6], which is a collection of C++ wrappers that encapsulate existing OS mechanisms for local and remote IPC such as sockets, `select()`, message queues, shared memory, semaphores, and remote procedure calls. The ACE C++ wrappers improve several software quality factors:

• **Correctness:** Wrappers improve the type-security of higher-level system programs and application code. For instance, applications need not access the weakly-typed, lower-level C system call interfaces directly. Instead, they may use the strongly-typed OO wrapper interfaces that detect type errors at compile-time rather than run-time. This, in turn, reduces a common source of subtle system programming errors.

• **Ease of Learning and Ease of Use:** Wrappers may be used to impose a hierarchical structure over an existing non-hierarchical API such as BSD sockets. Hierarchical APIs are typically easier to learn since their structure indicates closely related operations (such as client vs. server operations or local vs. remote operations). In addition, wrappers encapsulate many subtle and error-prone network programming details (such as network byte-ordering and host/service addressing) by (1) utilizing default values to simplify the interfaces for common usage patterns and (2) combining multiple functions that commonly occur together to form a single entry point. Providing simpler and more compact interfaces allows developers to concentrate on designing and implementing applications, instead of wrestling with the low-level details of the underlying networking code.

• **Portability:** Wrappers also improve application portability. For example, Figure 3 depicts how the `IPC_SAP` API (described in [7]) provides a portable veneer for many services offered in common by both BSD sockets and System V TLI. Application programs may then be written using a single object-oriented API, which is mapped transparently onto the appropriate system calls that access the particular underlying OS mechanisms. Note that a consistent API alone does not guarantee interoperability in an environment consisting of heterogeneous hosts and networks. In particular, interoperability is primarily a function of the underlying protocols and the semantics of the OS communication mechanisms. However, as shown in Figure 3, the core OS kernel protocol mechanisms such as TCP/IP *are* semantically compatible. Therefore, the primary interoperability problems arise from the lexical and syntactic diversity of the various APIs.

• **Extensibility:** When combined with C++ language features (such as inheritance, dynamic binding, and templates), the wrappers methodology also helps improve the extensibility of the existing OS interfaces. This is accomplished by carefully separating application *policies* from the networking and operating system *mechanisms*. The goal is to allow applications to extend the original APIs *without* modifying the design or implementation of the existing wrapper infrastructure [3].

This series of articles on C++ wrappers is intended to (1) increase software developers' knowledge of available UNIX IPC mechanisms, (2) describe an object-oriented methodology for designing and implementing distributed applications, and (3) present a strategy for organizations to migrate to OOD and C++. Many developers are not fluent with advanced UNIX IPC mechanisms such as System V IPC (*i.e.,* shared memory, message queues, and semaphores), the BSD socket interface to Internet- and UNIX-domain communication mechanisms, System V Transport Layer Interface (TLI), and Sun's RPC mechanisms. One reason for this general lack of fluency stems from the difficulty of understanding and using the advanced IPC mechanisms properly. Difficulties arise both from inadequate documentation, as well as from the complexity of the existing non-object-oriented IPC system call interfaces. For example, it is difficult to understand and use System V semaphores correctly since the API is quite general and sparsely documented [8]. The C++ wrapper semaphore interface developed for the ACE, on the other hand, shields developers from a myriad of unnecessary details. In particular, it is far more intuitive and simple to use the C++ wrapper version for applications that only require the standard *P* (wait) and *V* (signal) semaphore operations.

3 Integrating OOD and C++ into an Organization

Many software development organizations are planning to migrate their projects and products to use OOD and C++ in the near future. Their objective is typically to improve software quality factors such as portability, reusability, correctness, extensibility, and maintainability. However, there are several challenges to consider when making this transition. For example, designers and programmers must learn to use object-oriented methods and C++ effectively. The difficulty of this transition depends on the developers' prior level of expertise and the organization's commitment to training and education. Another challenge is to determine how to leverage off the large base of existing non-object-oriented library code and system software. For example, it is often impractical to rewrite all the existing code (which is often written in C) in C++. In addition, an organization must carefully select a suitable environment that enables portable development of C++ applications. This is particular important if multiple platforms (such as PCs and various workstation vendors) must be supported.

Based on my experience as a OOD/C++ developer and trainer during the past several years, I have found that devising C++ wrappers for existing OS mechanisms is an effective way to address all these challenges. In general, creating wrappers helps developers gain experience with OOD principles and techniques and C++ language features in an incremental, evolutionary manner. For example, a daunting challenge facing many novice object-oriented designers is determining how to decompose a particular problem domain into a suitable class hierarchy (*e.g.,* how to structure the linear BSD socket interface into a hierarchically structured object-oriented API). A wrapper-based training methodology helps to refine developers' skills by teaching them how to identify related classes of behavior and reorganize existing system components into more modular C++ class hierarchies.

In addition to helping developers gain confidence in their object-oriented abilities, C++ wrappers also leverage off the existing base of library code and systems software. By directly reusing implementations of non-object-oriented APIs and mechanisms, developers avoid becoming mired in complex, lower-level implementation details. Instead, they are free to concentrate on higher-level object-oriented design principles and class composition/decomposition issues.

Finally, developing C++ wrappers produces a suite of reusable abstractions that provide a solid foundation for subsequent projects. These projects may then use the evolving object-oriented infrastructure as the basis for writing applications entirely in C++. Moreover, if the wrappers are carefully designed and implemented, it is possible to port them between different C++ development environments with a minimal amount of effort.

4 Definitions and Future Directions

Table 1 defines key OS and networking terms used throughout the series of articles. The following is a list of topics that will be used to illustrate the C++ wrapper technique during upcoming issues of the C++ Report:

- Local and remote connection-oriented and connectionless IPC primitives such as socket-based and TLI-based network IPC
- I/O-based and timer-based event-driven port multiplexing and service dispatching mechanisms such as BSD `select()` and System V `poll()`
- Message-Oriented UNIX IPC mechanisms such as System V `message queues`, `named pipes`, and `STREAM pipes`
- Shared memory mechanisms such as System V *shared memory* and the BSD `mmap()` family of functions that support shared memory via a "memory-mapped file" abstraction.[1]
- System V Semaphores
- Remote Procedure Calls (RPCs)
- Explicit dynamic linking [9] facilities that enable the development of dynamically configured, multi-service network daemons
- The ADAPTIVE Service eXecutive (`ASX`), which provides a framework for developing and configuring concurrent stackable subservices
- The "distributed rwho" (`drwho`) integration framework, which extends and improves the functionality of the `rwhod` family of monitoring services (*e.g.,* `rwho`, `rusers`, and `ruptime`)

In general, appropriate candidates for C++ wrapper encapsulation are those OS mechanisms that are accessed via a "family" of system calls. In a certain sense, the existing system calls in a given family already represent the "member functions." However, using C++ wrappers helps to clarify the relationship between the various system calls.

[1] This abstraction exports the OS kernel's virtual memory mechanisms to application programs, allowing them to map files residing in secondary storage into one or more process address spaces.

5 Design Principles

Several general design principles guide the development of all the C++ wrappers presented in the subsequent articles:

1. Make it easy to use the underlying OS IPC mechanisms correctly, hard to use them incorrectly, but not impossible to use them in ways the class designers did not anticipate originally.
2. Provide developers with precisely the service interfaces they need, without burdening them with overly-general interfaces that contain many extraneous features their applications do not require. However, build the object-oriented API with extensibility and compatibility in mind.
3. Avoid adding extra performance overhead to the original C API. For example, the extra layer of abstraction added by C++ wrappers should not result in increased function call overhead. Developers are more likely to utilize the type-secure, well-structured OO abstractions if they do not impose any measurable performance overhead.

Accomplishing this latter performance goal involves the liberal use of inline functions. Since each member function in a wrapper class is generally quite short this does not lead to a significant amount of "code bloat." In addition, virtual functions are used sparingly in certain wrapper designs, since they typically incur additional run-time overhead. This is particularly important for wrappers such as `IPC_SAP` that provide only a "thin" OO veneer over the existing system call APIs. More sophisticated wrappers typically supply a sufficient amount of additional functionality that the amount of performance degradation may be insignificant by comparison.

6 Advantages and Disadvantages of C++

Each article also addresses certain advantages and disadvantages of using C++ as a systems programming language. While it is possible to develop portable wrappers for many of the IPC mechanisms described above using the C programming language, several distinct advantages accrue from using C++:

- Classes encourage modular APIs that simplify the development of portable and type-secure code.
- Parameterized types increase the reusability of the C++ class components.
- Inheritance and dynamic binding increases sharing, reusability, and extensibility of API components.

Term	Definition
API (Application Programmatic Interface)	An abstract interface used by applications to access the services and protocols provided by the underlying operating system and standard system libraries.
BSD Socket Layer	The API between user applications and the BSD networking subsystem in the OS kernel.
Daemon	An OS process that runs continuously "in the background" performing various services for clients.
Gateway	A hardware or software device that routes packets between networks.
Host	An addressable computer "end-system" attached to a network
Internetwork	A "network of potentially heterogeneous networks."
Internet	A network of interconnected networks using TCP/IP
Internet Protocol (IP)	A network layer (*i.e.*, OSI layer 3) protocol that performs segmentation, reassembly, and routing of packets.
IPC (Interprocess Communication)	A set of mechanisms that exchange data and control information between separate processes on local and/or remote hosts.
Network	A collection of host computers and the transmission media connecting the computers.
Operating System Kernel	A collection of basic operating system services (*e.g.,* process management, virtual memory, and low-level IPC).
PDU	A "protocol data unit," (often called a "packet") exchanged via IPC
Process	A program that is being executed by a host computer.
Protocol	A set of rules governing how two or more processes cooperate to exchange data.
Protocol Stack	An abstraction expressing hierarchical relations between protocols in one or more protocol suites
Protocol Suite	A collection of related protocols (*e.g.,* the Internet or OSI protocol suite).
`rwhod` Services	Programs utilizing the `rwhod` database to monitor and report remote host and user status.
Service	A collection of related operations that are exported to user applications and/or higher-layer services
Subnet	A logical or physical subcomponent of a larger network. Subnets are often created to simplify network administration, security, and/or routing.
Transmission Control Protocol	A connection-oriented transport protocol that reliably exchanges byte-streams of data *in-order* and *un-duplicated* between a local and remote process.
User Datagram Protocol (UDP)	An unreliable, connectionless transport protocol that exchanges datagrams between local and remote processes.

Table 1: Definitions for Common Networking and IPC Terms

- Default values simplify the calling interfaces for common API usage patterns.
- Inline functions eliminate overhead and enhance both abstraction and efficiency.
- The "external linkage" feature (*i.e.,* `extern "C"`) greatly improves inter-language interoperability and reuse of object code.

There are several disadvantages with using C++ (as it is currently defined). One is its lack of support for exception handling. This would be very useful for dealing with initialization failures in constructors. Without exception handling, it is difficult to propagate constructor failures up to an object's instantiation (since C++ constructors are not permitted to return error codes). A short-term workaround for the lack of exception handling is to (1) use default constructors that perform no work and (2) define and use `open()` member functions, which *do* return error codes if problems arise during initialization. However, this solution compromises type-security, since member functions may be called by mistake before the object is constructed properly. A widely-available, standardized exception handling mechanism should help this problem considerably.

Another annoying development problem is the amount of time that most C++ compilers require to instantiate templates. Template instantiation occurs at link-time and generally more than doubles or triples build time for large projects. Moreover, combining templates with shared libraries on certain OS platforms is rather tricky. A final problem is the lack of a standard language definition. When combined with the general lack of portability at the systems programming level, this make it difficult to easily port C++ code between OS platforms.

7 Summary

Developing C++ wrappers for existing OS IPC mechanisms helps to simplify the development of correct, concise, portable, and efficient distributed applications. C++ wrappers are also an effective method for introducing object-oriented design (OOD) and C++ into software development organizations. Subsequent articles will discuss examples of C++ wrappers in greater detail.

References

[1] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A Flexible and Adaptive Transport System Architecture to Support Lightweight Protocols for Multimedia Applications on High-Speed Networks," in *Proceedings of the* 1^{st} *Symposium on High-Performance Distributed Computing (HPDC-1)*, (Syracuse, New York), pp. 174–186, IEEE, September 1992.

[2] 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.

[3] 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.

[4] R. Campbell, V. Russo, and G. Johnson, "The Design of a Multiprocessor Operating System," in *Proceedings of the USENIX C++ Workshop*, pp. 109–126, USENIX Association, November 1987.

[5] D. C. Schmidt and T. Suda, "ADAPTIVE: A Framework for Experimenting with High-Performance Transport System Process Architectures," in *Proceedings of the* 2^{nd} *International Conference on Computer Communication Networks*, (San Diego, California), ISCA, June 1993.

[6] D. C. Schmidt, "The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software," in *Proceedings of the* 12^{th} *Annual Sun Users Group Conference*, (San Jose, CA), pp. 214–225, SUG, Dec. 1993.

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

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

[9] W. W. Ho and R. Olsson, "An Approach to Genuine Dynamic Linking," *Software: Practice and Experience*, vol. 21, pp. 375–390, Apr. 1991.