UNIVERSITY OF CALIFORNIA
Irvine

An Object-Oriented Framework for Experimenting with Alternative Process Architectures for Parallelizing Communication Subsystems

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Information and Computer Science

by

Douglas Craig Schmidt

Committee in charge:

Professor Tatsuya Suda, Chair

Professor Richard W. Selby, Co-Chair

Professor George Polyzos

1995

The dissertation of Douglas C. Schmidt is approved,

and is acceptable in quality and form for

publication on microfilm:

____

____

____

Committee Chair

University of California, Irvine

1995

Dedication

This dissertation is dedicated to the memory of Terry Williams (1971–1991). Terry was my friend and weight-lifting partner, who was killed in a tragic airplane accident in February, 1991. In the words of Marcus Antonius from Shakespeare's *Julius Caesar*:

"His life was gentle, and the elements so
Mixed in him that nature might stand up
And say to all the world: this was a man."

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my gratitude to my advisor, Professor Tatsuya Suda, for his guidance and support during my Ph.D. dissertation project. It has been very rewarding to have him as my advisor. My gratitude also goes to my co-advisor Professor Richard Selby, who has supported and counseled me throughout the course of my Ph.D. research. I would also like to thank Professor George Polyzos for serving on my thesis committee.

I would especially like to thank my girlfriend Christine D. Burgeson for her support and love during the past two years. She has been a constant source of inspiration, a wonderful dance partner, and a reluctant convert to viewing Arnold movies!

I would like to deeply thank my fellow UCI grad student friends: Dr. Adam Porter, Clark Savage Turner, and Dr. Kent Madsen. They filled my world with the sound of music and liberated me from the clutches of political correctness. Likewise, I would like to thank my weight-lifting partners: Dr. David Levine, Rich Mellon, and especially Terry Williams. They pumped up my endurance with their dedication to fitness.

I also gratefully acknowledge the contributions of Paul Stephenson of Ericsson/GE Mobile Communication. Paul dedicated countless hours to discuss techniques for developing object-oriented distributed system frameworks.

I would like to express my sincere appreciation to Dr. Dennis and Bea Volper, Dr. Steve Franklin, Dr. John King, and Dr. Jo Mahoney for their support and friendship during the time I have lived in Irvine, California.

I would also like to thank my research colleagues in Germany, Dr. Burkhard Stiller and Dr. Martina Zitterbart, for their friendship, encouragement, and earnest discussions during crucial stages of my Ph.D. project.

Finally, I would like to express very special thanks to my parents for their love, encouragement, endurance, and understanding during the past 14 years of my higher education experience.

Financial support has been provided by a number of sources throughout the years: NSF (NCR-8907909); University of California MICRO grants with Hitachi Ltd. and Hitachi America, NEC, Nippon Steel Information and Communication Systems Inc.

(ENICOM), Canon, Omron, Hughes Aircraft, and Hewlett Packard (HP); UCI Extension; Ericsson/GE Mobile Communications; and the Motorola IRIDIUM project.

Curriculum Vitae

1984	B.A. in Sociology, College of William and Mary, Williamsburg, VA
1986	M.A. in Sociology, College of William and Mary, Williamsburg, VA
	Thesis: *A Statistical Analysis of University Resource Allocation Policies.*
1990	M.S. in Information and Computer Science, University of California, Irvine
1994	Ph.D. in Information and Computer Science, University of California, Irvine
	Dissertation: *An Object-Oriented Framework for Experimenting with Alternative Process Architectures for Parallelizing Communication Subsystems.*

Publications

Refereed Academic Journal Publications

1. Douglas C. Schmidt and Tatsuya Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *Special issue on Configurable Distributed Systems in the Distributed Systems Engineering Journal*, BCS/IEE, January, 1995.
2. Douglas C. Schmidt, Donald F. Box, and Tatsuya Suda, "ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment," *Journal of Concurrency: Practice and Experience*, Wiley and Sons, Ltd., Vol. 5, No. 4, June, 1993, pp. 269–286.
3. Douglas C. Schmidt and Tatsuya Suda, "Transport System Architecture Services for High-Performance Communication Systems," *Journal of Selected Areas of Communications special-issue on Protocols for Gigabit Networks*, IEEE, Vol. 11, No. 4, May, 1993, pp. 489–506.

Refereed Academic Conference Publications

4. "Performance Experiments on Alternative Methods for Structuring Active Objects in High-Performance Parallel Communication Systems," 9^{th} *OOPSLA Conference, poster session*, ACM, Portland, Oregon, October, 1994.
5. Douglas C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching," *Proceedings of the* 1^{st} *Annual Conference on the Pattern Languages of Programs*, Monticello, Illinois, August, 1994.
6. Douglas C. Schmidt and Tatsuya Suda, "Experiences with an Object-Oriented Architecture for Developing Dynamically Extensible Network Management Software," *Proceedings of the Globecom Conference*, IEEE, San Francisco, California, November, 1994.
7. Douglas C. Schmidt, Burkhard Stiller, Tatsuya Suda, and Martina Zitterbart, "Configuring Function-based Communication Protocols for Distributed Applications," *Proceedings of the* 8^{th} *International Working Conference on Upper Layer Protocols, Architectures, and Applications*, IFIP, Barcelona, Spain, June 1-3, 1994, pp. 1–13.
8. Douglas C. Schmidt and Tatsuya Suda, "The ADAPTIVE Service Executive: An Object-Oriented Architecture for Configuring Concurrent Distributed Communication Systems," *Proceedings of the* 8^{th} *International Working Conference on Upper Layer Protocols, Architectures, and Applications*, IFIP, Barcelona, Spain, June 1-3, 1994, pp. 1–14.
9. Douglas C. Schmidt, "ASX: An Object-Oriented Framework for Developing Distributed Applications," *Proceedings of the* 6^{th} *C++ Conference*, USENIX, Cambridge, Massachusetts, April, 1994.
10. Douglas C. Schmidt, Burkhard Stiller, Tatsuya Suda, Ahmed Tantawy, and Martina Zitterbart, "Configuration Support for Flexible Function-Based Communication Systems," *Proceedings of the* 18^{th} *Conference on Local Computer Networks*, IEEE, Minneapolis, Minnesota, September 20-22, 1993, pp. 369–378.
11. Douglas C. Schmidt and Tatsuya Suda, "ADAPTIVE: a Framework for Experimenting with High-Performance Transport System Process Architectures," *Proceedings of the* 2^{nd} *International Conference on Computer Communications and Networks*, ISCA, San Diego, California, June 28-30, 1993, pp. 1–8.
12. Donald F. Box, Douglas C. Schmidt, and Tatsuya Suda, "ADAPTIVE: An Object-Oriented Framework for Flexible and Adaptive Communication Protocols," *Proceedings of the* 4^{th} *Conference on High Performance Networking*, IFIP, Liege, Belgium, December 14-18, 1992, pp. 367–382.
13. Douglas C. Schmidt, Donald F. Box, and Tatsuya Suda, "ADAPTIVE: A Flexible and Adaptive Transport System Architecture to Support Lightweight Protocols

for Multimedia Applications on High-Speed Networks," *Proceedings of the* 1^{st} *Symposium on High Performance Distributed Computing*, IEEE, Syracuse, New York, September 9-11, 1992, pp. 174–186.

14. Richard W. Selby, Adam A. Porter, Douglas C. Schmidt, and James Berney, "Metric-Driven Analysis and Feedback Systems for Enabling Empirically Guided Software Development," *Proceedings of the* 13^{th} *Annual International Conference on Software Engineering*, IEEE, Austin, Texas, May, 1991, pp. 430–443.
15. Douglas C. Schmidt "GPERF: A Perfect Hash Function Generator," *Proceedings of the* 2^{nd} *C++ Conference*, USENIX, San Francisco, California, April 9-11, 1990, pp. 87–102.

Refereed Academic Workshop Publications

16. Douglas C. Schmidt and Tatsuya Suda, "Measuring the Impact of Alternative Parallel Process Architectures on Communication Subsystem Performance," *Proceedings of the Proceedings of the* 4^{th} *International Workshop on Protocols for High-Speed Networks*, IFIP, Vancouver, British Columbia, August, 1994, pp. 1–17.
17. Douglas C. Schmidt and Tatsuya Suda, "The Service Configurator Framework: An Extensible Architecture for Dynamically Configuring Concurrent, Multi-service Network Daemons," *Proceedings of the* 2^{nd} *International Workshop on Configurable Distributed Systems*, IEEE, Pittsburgh, PA, March 21-23, 1994, pp. 190–201.
18. Douglas C. Schmidt, Burkhard Stiller, Tatsuya Suda, and Martina Zitterbart, "Tools for Generating Application-Tailored Multimedia Protocols on Heterogeneous Multi-Processor Platforms," *Proceedings of the* 2^{nd} *Workshop on High-Performance Communications Subsystems*, IEEE, Williamsburg, Virginia, September 1-3, 1993, pp. 1–7.
19. Douglas C. Schmidt and Tatsuya Suda, "A Framework for Developing and Experimenting with Parallel Process Architectures to Support High-Performance Transport Systems," *Proceedings of the* 2^{nd} *Workshop on High-Performance Communications Subsystems*, IEEE, Williamsburg, Virginia, September 1-3, 1993, pp. 1–8.
20. Tatsuya Suda, Douglas C. Schmidt, Donald F. Box, Duke Hong and Hung Huang, "High Speed Networks," *Proceedings of the International Computer World Symposium '92*, Kobe, Japan, November, 1992.
21. Hung K. Huang, Douglas C. Schmidt, Donald F. Box, Kazu Shimono, Girish Kotmire, Unmesh Rathi, and Tatsuya Suda, "ADAPTIVE: A Prototyping Environment for Transport Systems." *Proceedings of the* 4^{th} *International Workshop on*

Computer Aided Modeling, Analysis, and Design of Communication Links and Networks (CAMAD), IEEE, Montreal, Canada, September, 1992.

22. Donald F. Box, Douglas C. Schmidt, and Tatsuya Suda, "Alternative Approaches to ATM/Internet Interoperation," *Proceedings of the* 1^{st} *Workshop on the Architecture and Implementation of High-Performance Communication Subsystems*, IEEE, Tucson, Arizona, February 17-19, 1992, pp. 1–5.

23. Douglas C. Schmidt and Richard Selby "Modeling Software Interconnectivity," *Proceedings of the* 22^{nd} *Symposium on the Interface: Computer Science and Statistics*, East Lansing, MI, May, 1990.

24. Richard W. Selby, Greg James, Kent Madsen, Joan Mahoney, Adam A. Porter, and Douglas C. Schmidt "Classification Tree Analysis Using the Amadeus Measurement and Empirical Analysis System," *Proceedings of the* 14^{th} *Annual Software Engineering Workshop at NASA Software Engineering Laboratory*, College Park, Maryland, November, 1989, pp. 239–250.

Refereed Trade Journal Publications

25. Douglas C. Schmidt, "Transparently Parameterizing Synchronization Mechanisms into a Concurrent Distributed Application," *C++ Report*, SIGS, Vol. 6, No. 5, July/August 1994, pp. 1–10.

26. Douglas C. Schmidt, "A Domain Analysis of Network Daemon Design Dimensions," *C++ Report*, SIGS, Vol. 6, No. 3, March/April, 1994, pp. 1–12.

27. Douglas C. Schmidt, "The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing," *C++ Report*, SIGS, Vol. 5, No. 7, September, 1993, pp. 1–14.

28. Douglas C. Schmidt, "The Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing," *C++ Report*, SIGS, Vol. 5, No. 2, February, 1993, pp. 1–12.

29. Douglas C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Operating System Interprocess Communication Services," *C++ Report*, SIGS, Vol. 4, No. 8, November/December, 1992, pp. 1–10.

30. Douglas C. Schmidt, "Systems Programming with C++ Wrappers: Encapsulating Interprocess Communication Services with Object-Oriented Interfaces," *C++ Report*, SIGS, Vol. 4, No. 7, September/October, 1992, pp 1–6.

Refereed Trade Conference Publications

31. Douglas C. Schmidt and Paul Stephenson, "Achieving Reuse Through Design Patterns," *Proceedings of the* 3^{rd} *Annual C++ World Conference*, SIGS, Austin, Texas, November 14-18, 1994.

32. Douglas C. Schmidt, "The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Distributed Applications," *Proceedings of the* 12^{th} *Annual Sun Users Group Conference*, SUG, San Francisco, June 16-17, 1994. This paper won the "best student paper" award at the conference.

33. Douglas C. Schmidt, "The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications," *Proceedings of the* 11^{th} *Annual Sun Users Group Conference*, SUG, San Jose, December 7-9, 1993, pp. 214–225. This paper won the "best student paper" award at the conference.

34. Douglas C. Schmidt and Paul Stephenson, "An Object-Oriented Framework for Developing Network Server Daemons," *Proceedings of the* 2^{nd} *Annual C++ World Conference*, SIGS, Dallas, Texas, October 18-22, 1993, pp. 73–85.

Honors and Awards

- Invited to join the faculty at Washington University, in St. Louis, Missouri as an assistant faculty member from August, 1994 to present.
- Selected to participate in the ACM OOPSLA '94 Doctoral Symposium.
- Invited by Dr. Martina Zitterbart to participate in a 4-week international exchange program at the Universität Karlsruhe Institut für Telematik in Karlsruhe, Germany, April, 1993.
- Invited to write a column on distributed object computing for the *C++ Report* magazine from July, 1994 to present.
- Invited contributor to the *C++ Report* magazine from July 1992 to present.
- Served as elected representative to the Associated Graduate Student organization at the University of California, Irvine from May, 1991 to June, 1992.
- Served as elected graduate student representative to the Computer Science Computing Resource Committee at the University of California, Irvine from August, 1988 to August, 1990.
- Invited to work with Dr. Peter G. W. Keen at the International Center for Information Technology, Washington D.C. on a project assessing techniques for improving software productivity in the summer of 1987.

- Awarded Teaching and Research Assistantships in Computer Science at University of California, Irvine during 1986-1994.
- Awarded Research Assistantship in Sociology at the College of William and Mary during 1985-1986.

Fields of Study

Distributed systems

Parallel processing

High-performance communication subsystems and protocols

Object-oriented design and programming

Abstract of the Dissertation

An Object-Oriented Framework for Experimenting with Alternative Process Architectures for Parallelizing Communication Subsystems

by

Douglas C. Schmidt

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1995

Professor Tatsuya Suda, Chair

Professor Richard Selby, Co-chair

The demand for high-performance distributed communication systems (such as video-on-demand servers, global personal communication systems, and the underlying communication protocol stacks) is increasing dramatically. Distributing communication services throughout high-speed computer networks offers many potential benefits by increasing performance, scalability, and functionality. In particular, performing communication services in parallel helps to improve performance by increasing processing rates and reducing latency. To improve performance significantly, however, the speed-up obtained from parallel processing must outweigh the major sources of overhead associated with parallel processing. On multi-processor platforms based on shared memory (rather

than message passing), these sources of overhead primarily involve context switching, synchronization, and data movement.

Many communication systems (such as the layered protocol stacks specified by the TCP/IP and the ISO OSI reference models) decompose naturally into a series of hierarchically-related tasks. A number of process architectures have been proposed as the basis for parallelizing these types of communication systems in order to improve performance. There are two fundamental types of process architectures: task-based and message-based. Task-based process architectures are formed by binding one or more processing elements to the layers of tasks in a communication system. In contrast, message-based process architectures are formed by binding the processing elements to the data messages and control messages that flow through the layers of tasks. Each type of process architecture incurs different levels of context switching, synchronization, and data movement overhead. This overhead is affected by factors such as the application requirements, OS and hardware platform, and network characteristics.

This dissertation describes parallel process architecture performance experiments conducted using the ADAPTIVE Service eXecutive (`ASX`) framework. The purpose of this research is to identify architectures for structuring parallelism to reduce the overhead incurred on shared memory multi-processing platforms. The `ASX` framework facilitates the flexible configuration of high-performance distributed communication systems that effectively utilize parallelism on shared memory multi-processor platforms. The `ASX` framework controls for a number of relevant confounding factors (such as application and protocol functionality, concurrency control schemes, and application traffic characteristics). By controlling these factors, the `ASX` framework enables precise measurement of the performance impact of alternative process architectures for parallelizing communication protocol stacks. The dissertation describes the object-oriented architecture of the `ASX` framework and presents results from the process architecture performance experiments.

Chapter 1

Introduction

Advances in VLSI and fiber optic technology are shifting performance bottlenecks from the underlying networks to the communication subsystem. A communication subsystem consists of *protocol tasks* and *operating system mechanisms*. Protocol tasks include connection establishment and termination, end-to-end flow control, remote context management, segmentation/reassembly, demultiplexing, error protection, session control, and presentation conversions. Operating system mechanisms include process management, timer-based and I/O-based event invocation, message buffering, and layer-to-layer flow control. Together, protocol tasks and operating system mechanisms support the implementation and execution of communication protocol stacks composed of protocol tasks [SS93].

Executing protocol stacks in parallel on multi-processor platforms is a promising technique for increasing protocol processing performance [ZST93]. Significant increases in performance are possible, however, only if the speed-up obtained from parallelism outweights the context switching and synchronization overhead associated with parallel processing. A context switch is triggered when an executing process relinquishes its associated processing element (PE) voluntarily or involuntarily. Depending on the underlying OS and hardware platform, a context switch may require dozens to hundreds of instructions to flush register windows, memory caches, instruction pipelines, and translation look-aside buffers [MB91]. Synchronization overhead arises from locking mechanisms that serialize access to shared objects (such as message buffers, message queues, protocol connection records, and demultiplexing maps) used during protocol processing [Mat93].

A number of *process architectures* have been proposed as the basis for parallelizing communication subsystems [Mat93, Zit91, GNI92, PS93, JSB90]. There are two fundamental types of process architectures: *task-based* and *message-based*. Task-based process architectures are formed by binding one or more PEs to units of protocol functionality (such as presentation layer formatting, transport layer end-to-end flow control, and network layer fragmentation and reassembly). In this architecture, parallelism is achieved by executing protocol tasks in separate PEs, and passing data messages and control messages between the tasks/PEs. In contrast, message-based process architectures are formed by binding the PEs to data messages and control messages received from applications and network interfaces. In this architecture, parallelism is achieved by simultaneously escorting multiple data messages and control messages on separate PEs through a stack of protocol tasks.

Protocol stacks (such as the TCP/IP protocol stack and the ISO OSI 7 layer protocol stack) may be implemented using either task-based or message-based process architectures. However, these two types of process architectures exhibit significantly different performance characteristics that vary across operating system and hardware platforms. For instance, on shared memory multi-processor platforms, task-based process architectures exhibit high context switching and data movement overhead due to scheduling and caching properties of the OS and hardware [WF93]. In contrast, in a message-passing multi-processor environment, message-based process architectures exhibit high levels of synchronization overhead due to high latency access to global resources such as shared memory, synchronization objects, or connection context information [Zit91].

Existing studies have generally selected a single task-based or message-based process architecture and studied it in isolation. Moreover, these studies have been conducted on different OS and hardware platforms, using different protocol stacks and implementation techniques, which makes it difficult to meaningfully compare results. In this paper, we describe the design and implementation of an object-oriented framework that supports controlled experiments with several alternative parallel process architectures. The framework controls for a number of relevant confounding factors (such as protocol functionality, concurrency control strategies, application traffic characteristics, and network interfaces), which enables precise measurement of the performance impact of using different process architectures to parallelize communication protocol stacks. This paper reports the results of systematic, empirical comparisons of the performance of several message-based and task-based process architectures implemented on a widely-available shared memory multi-processor platform.

The organization of this dissertation is as follows. Chapter 2 presents a survey of software mechanisms that comprise the architecture of communication subsystems. This chapter also outlines the two fundamental types of process architectures and classifies related work accordingly.

Chapter 3 describes the structure and functionality of the ADAPTIVE Service eXecutive `ASX` framework. The `ASX` framework provides an integrated set of object-oriented components that facilitate the development of, and experimentation with, parallel process architectures on multi-processor platforms.

Chapter 4 examines empirical results from parallel process architecture experiments conducted using the `ASX` framework. These experiments demonstrate the extent to which different process architectures affect protocol stack performance.

Chapter 5 presents concluding remarks and outlines future research directions.

Chapter 2

A Survey of Software Components in Communication Subsystem Architectures

2.1 Introduction

The demand for many types of distributed applications is expanding rapidly, and application requirements and usage patterns are undergoing significant changes. When coupled with the increased channel speeds and services offered by high-performance networks, these changes make it difficult for existing communication subsystems to process application data at network channel speeds. This chapter examines communication subsystem mechanisms that support bandwidth-intensive multimedia applications such as medical imaging, scientific visualization, full-motion video, and tele-conferencing. These applications possess quality-of-service requirements that are significantly different from conventional data-oriented applications such as remote login, email, and file transfer.

Multimedia applications involve combinations of requirements such as extremely high throughput (full-motion video), strict real-time delivery (manufacturing control systems), low latency (on-line transaction processing), low delay jitter (voice conversation), capabilities for multicast (collaborative work activities) and broadcast (distributed name resolution), high-reliability (medical image transfer), temporal synchronization (tele-conferencing), and some degree of loss tolerance (hierarchically-coded video). Applications also impose different network traffic patterns. For instance, some applications generate highly bursty traffic (variable bit-rate video), some generate continuous traffic (constant bit-rate video), and others generate short-duration, interactive, request-response traffic (network file systems using remote procedure calls (RPC)).

Application performance is affected by a variety of network and communication subsystem factors. Networks provide a transmission framework for exchanging various types of information (such as voice, video, text, and images) between gateways, bridges, and hosts. Example networks include the Fiber Distributed Data Interface (FDDI), the Distributed Queue Dual Bus (DQDB), the Asynchronous Transfer Mode (ATM), X.25 networks, and IEEE 802 LANs. In general, the lower-layer, link-to-link network protocols are implemented in hardware.

Communication subsystems integrate higher-layer, end-to-end communication protocols such as TCP, TP4, VMTP, XTP, RPC/XDR, and ASN.1/BER together with the operating system (OS) mechanisms provided by end systems. The tasks performed by the communication subsystem may be classified into several levels of abstraction. The highest level provides an application interface that mediates access to end-to-end communication protocols. These protocols represent an intermediate level of abstraction that provides presentation and transport mechanisms for various connectionless, connection-oriented, and request-response protocols. These mechanisms are implemented via protocol tasks such as connection management, flow control, error detection, retransmission, encryption, and compression schemes. Both the application interface and the protocols operate within an OS framework that orchestrates various hardware resources (*e.g.,* CPU(s), primary and secondary storage, and network adapters) and software components (*e.g.,* virtual memory, process architectures, message managers, and protocol graphs) to support the execution of distributed applications.

Performance bottlenecks are shifting from the underlying networks to the communication subsystem. This shift is occurring due to advances in VLSI technology and fiber optic transmission techniques that have increased network channel speeds by several orders of magnitude. Increasing channel speeds accentuate certain sources of communication subsystem overhead such as memory-to-memory copying and process management operations like context switching and scheduling. This mismatch between the performance of networks and the communication subsystem constitutes a *throughput preservation problem* [MS92], where only a portion of the available network bandwidth is actually delivered to applications on an end-to-end basis.

In general, sources of communication subsystem overhead are not decreasing as rapidly as network channel speeds are increasing. This results from factors such as improperly layered communication subsystem architectures [CWWS92, HP91]. It is also exacerbated by the widespread use of operating systems that are not well-suited to asynchronous, interrupt-driven network communication. For example, many network adapters generate interrupts for every transmitted and received packet, which increases the number of CPU context switches [Haa91, KC88]. Despite increasing total MIPS, RISC-based computer architectures exhibiting high context switching overhead that penalizes interrupt-driven network communication. This overhead results from the cost of flushing pipelines, invalidating CPU instruction/data caches and virtual memory translation-lookaside buffers, and managing register windows [JSB90].

Alleviating the throughput preservation problem and providing very high data rates to applications requires the modification of conventional communication subsystem architectures [SP90]. To help system researchers navigate through the communication subsystem design space, this chapter presents a taxonomy of six key communication subsystem mechanisms including the process architecture, virtual remapping, and event management dimensions, as well as the message management, multiplexing and demultiplexing, and layer-to-layer flow control dimensions. The taxonomy is used to compare and contrast four general-purpose commercial and experimental communication subsystems (System V STREAMS [Rag93], the BSD UNIX network subsystem [LMKQ89], the *x*-kernel [HP91], and the Conduit framework from the Choices operating system [Zwe90]). The intent of the chapter is to explore communication subsystem design alternatives that support distributed applications effectively.

This chapter is organized as follows: Section 2.2 outlines the general architectural components in a communication subsystem; Section 2.3 describes a taxonomy for classifying communication subsystems according to their kernel and protocol family architecture dimensions; Section 2.4 provides a comparison of four representative communication subsystems; and Section 2.5 presents concluding remarks.

2.2 Levels of Abstraction in a Communication Subsystem Architecture

Communication subsystem architectures provide a framework for implementing end-to-end protocols that support distributed applications operating over local and wide area networks. This framework integrates hardware resources and software components used to implement *protocol graphs* [OP92]. A protocol graph characterizes hierarchical

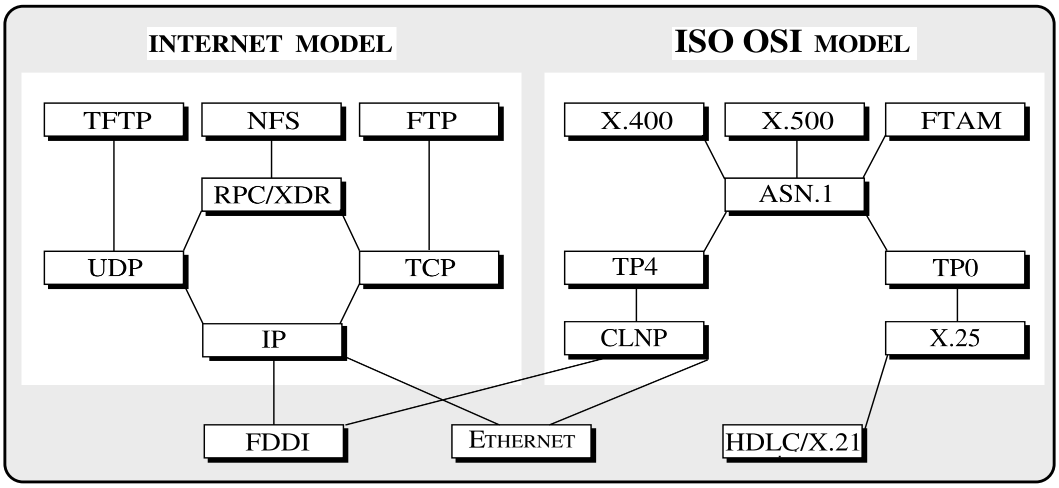

Figure 2.1: Protocol Graph for Internet and OSI Communication Models

relations between protocols in communication models such as the Internet, OSI, XNS, and SNA. Figure 2.1 depicts protocol graphs for the Internet and OSI communication models. Each node in a protocol graph represents a protocol such as RPC/XDR, TCP, IP, TP4, or CLNP.

Protocol graphs are implemented via mechanisms provided by the communication subsystem architecture. Communication subsystems may be modeled as nested virtual machines that constitute different levels of abstraction. Each level of virtual machine is characterized by the mechanisms it exports to the surrounding levels. The model depicted in Figure 2.2 represents an abstraction of hardware and software mechanisms found in conventional communication subsystems. Although certain communication subsystems bypass or combine adjacent levels for performance reasons [CT90, Ten89], Figure 2.2 provides a concise model of the relationships between major communication subsystem components.

The hierarchical relationships illustrated by the protocol graph in Figure 2.1 are generally orthogonal to the levels of abstraction depicted by the communication subsystem virtual machines shown in Figure 2.2. In particular, protocol graphs in Figure 2.1 are

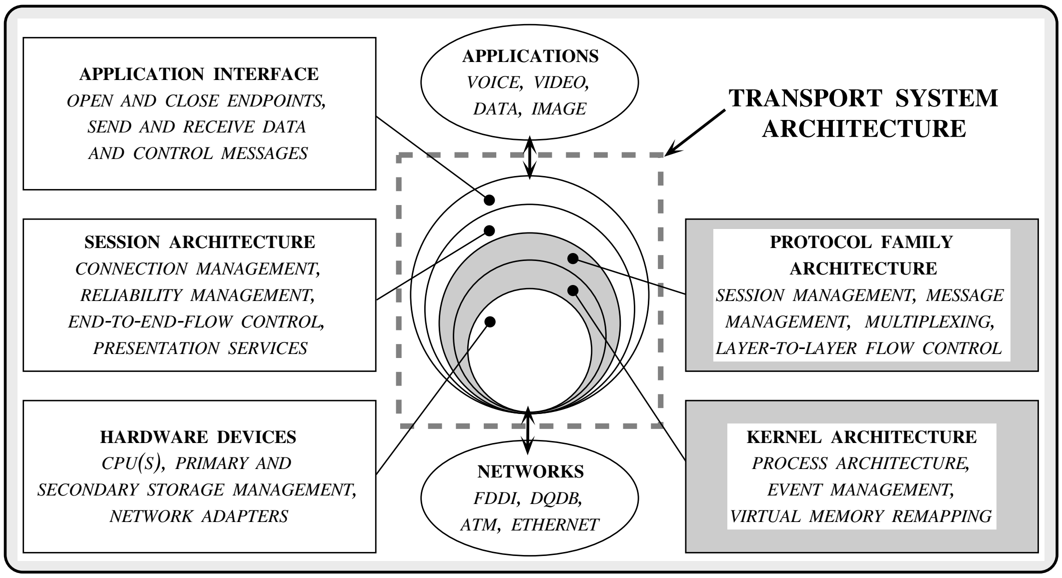

Figure 2.2: Architectural Components in a Communication Subsystem

implemented via the communication subsystem mechanisms shown in Figure 2.2. The following paragraphs summarize the key levels in the communication subsystem, which consist of the *application interface*, *session architecture*, *protocol family architecture*, and *kernel architecture*.

As shown by the shaded portions of Figure 2.2, this chapter focuses on the kernel architecture (described in Section 2.3.1) and the protocol family architecture (described in Section 2.3.2). A thorough discussion of the application interface is beyond the scope of this chapter and topics involving the session architecture are discussed further in [SSS+93]. These two components are briefly outlined below for completeness and to provide a context for discussing the other levels.

2.2.1 Application Interface

The *application interface* is the outermost-level of a communication subsystem. Since protocol software often resides within the protected address space of an operating system kernel, programs utilize this application interface to interact with inner-level communication subsystem mechanisms. The application interface transfers data and control information between user processes and the session architecture mechanisms that perform connection management, option negotiation, data transmission control, and error protection. BSD UNIX sockets [LMKQ89] and System V UNIX TLI [Sun92] are widely available examples of application interfaces.

Performance measurements indicate that conventional application interfaces constitute 30 to 40 percent of the overall communication subsystem overhead [HP91, ACR88]. Much of this overhead results from the memory-to-memory copying and process synchronization that occurs between application programs and the inner-level communication subsystem mechanisms. The functionality and performance of several application interfaces is evaluated in [Che87, MK91].

2.2.2 Session Architecture

The next level of the communication subsystem is the *session architecture*, which performs "end-to-end" network tasks. Session architecture mechanisms are associated with local end-points of network communication, often referred to as protocol *sessions*.[1] A session consists of data structures that store context information and subroutines that implement the end-to-end protocol state machine operations.

Session architecture mechanisms help satisfy end-to-end application quality-of-service requirements involving throughput, latency, and reliability [JBB92]. In particular, quality-of-service is affected by session architecture mechanisms that manage *connections* (*e.g.,* opening and closing end-to-end network connections, and reporting

[1]The term "session" is used in this chapter in a manner not equivalent to the ISO OSI term "session layer."

and updating connection context information), *reliability* (*e.g.,* computing checksums, detecting mis-sequenced or duplicated messages, and performing acknowledgments and retransmissions), and *end-to-end flow and congestion* (*e.g.,* advertizing available window sizes and tracking round-trip packet delays). In addition, session architecture mechanisms also manage per-connection *protocol interpreters* (*e.g.,* controlling transitions in a transport protocol's state machine) and *presentation services* (*e.g.,* encryption, compression, and network byte-ordering conversions). Various session architecture issues are examined in [PS91, DDK$^+$90, Svo89, SSS$^+$93].

2.2.3 Protocol Family Architecture

The *protocol family architecture*[2] provides *intra-* and *inter-protocol* mechanisms that operate within and between nodes in a protocol graph, respectively. Intra-protocol mechanisms manage the creation and destruction of sessions that are managed by the session architecture described above. Inter-protocol mechanisms provide message management, multiplexing and demultiplexing, and layer-to-layer flow control.

The primary difference between the session architecture and the protocol family architecture is that session architecture mechanisms manage the *end-to-end* processing activities for network connections, whereas protocol family architecture mechanisms manage the *layer-to-layer* processing activities that occur within multi-layer protocol graphs. In some cases, these activities are entirely different (*e.g.,* the presentation services provided by the session architecture such as encryption, compression, and network

[2] A protocol family is a collection of network protocols that share related *communications syntax* (*e.g.,* addressing formats), *semantics* (*e.g.,* interpretation of standard control messages), and *operations* (*e.g.,* demultiplexing schemes and checksum computation algorithms). A wide range of protocol families exist such as SNA, TCP/IP, XNS, and OSI.

byte-ordering are unnecessary in the protocol family architecture). In other cases, different mechanisms are used to implement the same abstract task.

The latter point is exemplified by examining several mechanisms commonly used to implement flow control. *End-to-end* flow control is a session architecture mechanism that employs sliding window or rate control schemes to synchronize the amount of data exchanged between sender(s) and receiver(s) communicating at the same protocol layer (*e.g.,* between two TCP connection end-points residing on different hosts). *Layer-to-layer* flow control, on the other hand, is a protocol family architecture mechanism that regulates the amount of data exchanged between adjacent layers in a protocol graph (*e.g.,* between the TCP and IP STREAM modules in System V STREAMS). In general, end-to-end flow control requires distributed context information, whereas layer-to-layer flow control does not.

Mechanisms in the protocol family architecture are often reusable across a wide-range of communication protocols. In contrast, session architecture mechanisms tend to be reusable mostly within a particular class of protocols. For instance, most communication protocols require some form of message buffering support (which is a protocol family architecture mechanism). However, not all communication protocols require retransmission, flow control, or connection management support. In addition, certain protocols may only work with specific session architecture mechanisms (such as the standard TCP specification that requires sliding-window flow control and cumulative acknowledgment).

2.2.4 Kernel Architecture

The *kernel architecture*[3] provides mechanisms that manage hardware resources such as CPU(s), primary and secondary storage, and various I/O devices and network adapters. These mechanisms support concurrent execution of multiple protocol tasks on uni- and multi-processors, virtual memory management, and event handling. It is crucial to implement kernel architecture mechanisms efficiently since the application interface and session and protocol family architectures ultimately operate by using these mechanisms. The primary distinction between the protocol family architecture and the kernel architecture is that kernel mechanisms are also utilized by user applications and other OS subsystems such as the graphical user interface or file subsystems. In contrast, protocol family architecture mechanisms are concerned primarily with the communication subsystem.

2.3 A Taxonomy of Communication Subsystem Architecture Mechanisms

Table 2.1 presents a taxonomy of six key kernel architecture and protocol family architecture mechanisms that support the layer-to-layer computing requirements of protocol graphs end systems. The following section describes the communication subsystem mechanisms presented in Table 2.1.

[3]The term "kernel architecture" is used within this chapter to identify mechanisms that form the "nucleus" of the communication subsystem. However, protocol and session architecture components may reside within an OS kernel (BSD UNIX [LMKQ89], and System V UNIX [Rag93]), in user-space (Mach [ABG+86] and the Conduit framework [Zwe90]), in either location (the *x*-kernel [HP91]), or in off-board processors (Nectar [CSSZ90] and VMP [KC88]).

Category	Dimension	Subdimension	Alternatives
Kernel Architecture Dimensions	Process Architecture	(1) Concurrency Models	single-threaded, HWP, LWP, coroutines
		(2) Process Architectures	message-based, task-based, hybrid
	VM Remapping		outgoing and/or incoming
	Event Management	(1) Search Structure	array, linked list, heap
		(2) Time Relationships	relative, absolute
Protocol Family Architecture Dimensions	Message Management	(1) Memory Management	uniform, non-uniform performance
		(2) Memory Copy Avoidance	list-based, DAG-based data structure
	Muxing and Demuxing	(1) Synchronization	synchronous, asynchronous
		(2) Layering	layered, de-layered
		(3) Searching	indexing, sequential search, hashing
		(4) Caching	single-item, multiple-item
	Layer-to-layer Flow Control		per-queue, per-process

Table 2.1: Communication Subsystem Taxonomy Template

2.3.1 Kernel Architecture Dimensions

As described below, the kernel architecture provides the *process architecture*, *virtual memory remapping*, and *event management* mechanisms utilized by the session and protocol family architectures.

2.3.1.1 The Process Architecture Dimension

A process is a collection of resources (such as file descriptors, signal handlers, a run-time stack, etc.) that may support the execution of instructions within an address space. This address space may be shared with other processes. Other terms (such as threads [TRG$^+$87] or light-weight processes [EKB$^+$92]) are often used to denote the same basic concept. Our use of the term process is consistent with the definition adopted in [Pre93, Gar90].

A process architecture represents a binding between various units of communication protocol processing (such as layers, functions, connections, and messages) and various structural configurations of processes. The process architecture selected for a communication subsystem is one of several factors (along with protocol designs/implementations and bus, memory, and network interface characteristics) that impact overall application performance. In addition, the choice of process architecture also influences demultiplexing strategies [Fel90] and protocol programming techniques [HP91, Atk88].

Several concurrency models are outlined below. These models form the basis for implementing the alternative process architectures that are examined in detail following concurrency model discussion. In order to produce efficient communication subsystems, it is important to match the selected process architecture with the appropriate concurrency model.

(1) Concurrency Models: *Heavy-weight processes*, *light-weight processes*, and *coroutines* are concurrency models used to implement process architectures. Each model exhibits different performance characteristics and allows different levels of control over process management activities such as scheduling and synchronization. The following paragraphs describe key characteristics of each concurrency model:

- **Heavy-Weight Processes:** A heavy-weight process (HWP) typically resides in a separate virtual address space managed by the OS kernel and the hardware memory management unit. Synchronizing, scheduling, and exchanging messages between HWPs involves context switching, which is a relatively expensive operation in many operating systems. Therefore, HWPs may not be an appropriate choice for executing multiple interacting protocol processing activities concurrently.

• **Light-Weight Processes:** Light-weight processes (LWPs) differ from HWPs since multiple LWPs generally *share* an address space by default. This sharing reduces the overhead of LWP creation, scheduling, synchronization, and communication for the following reasons:

- Context switching between LWPs is less time consuming than HWPs since there is less context information to store and retrieve
- It may not be necessary to perform a "mode switch" between kernel- and user-mode when scheduling and executing an LWP [EKB+92]
- Communication between LWPs may use shared memory rather than message passing

Note that LWPs may be implemented in kernel-space, user-space, or some hybrid configuration [ABLL92].

• **Coroutines:** In the coroutine model, a developer (rather than an OS scheduler) explicitly chooses the next coroutine to run at a particular synchronization point. When a synchronization point is reached, the coroutine suspends its activities to allow another coroutine to execute. At some later point, the second coroutine may resume control back to the first coroutine. Coroutines provide developers with the flexibility to schedule and execute tasks in any desired manner. However, developers also assume responsibility for handling all scheduling details, as well as avoiding starvation and deadlock.

Executing protocol and session mechanisms via multiple processes is often less complicated and error-prone than synchronizing and scheduling these mechanisms manually via coroutines. In addition, coroutines support only interleaved process execution, which limits the benefits of multi-processing since only one process may run at any given

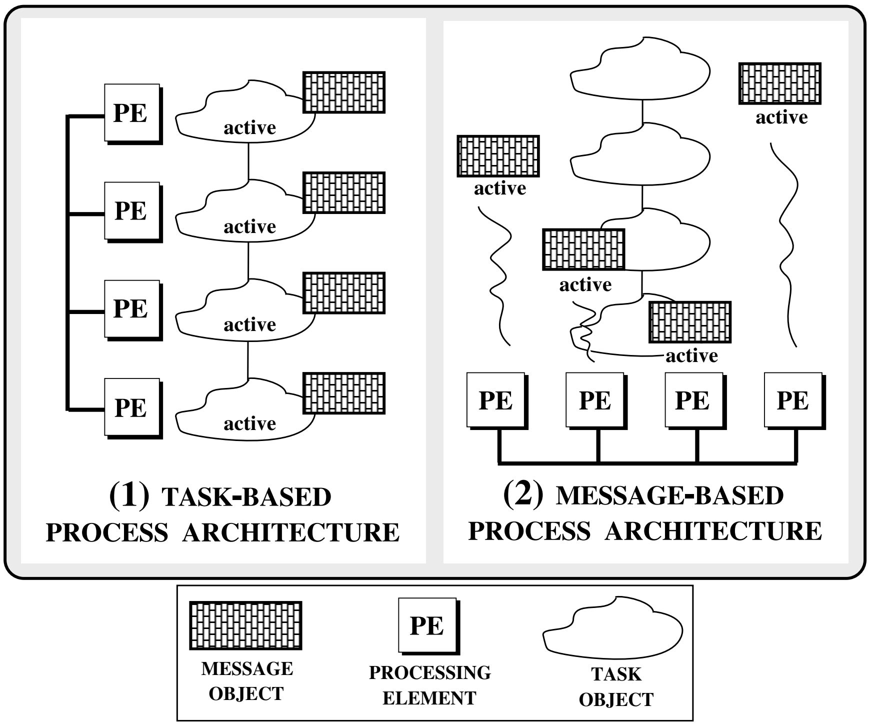

Figure 2.3: Process Architecture Components and Interrelationships

time. In general, it appears that LWPs are a more appropriate mechanism for implementing process architectures than HWPs since minimizing context switching overhead is essential for high-performance [HP91]. Even with LWPs, however, to it is still important to perform concurrent processing efficiently to reduce the overhead from (1) preempting, rescheduling, and synchronizing executing processes and (2) serializing access to shared resources must be minimized.

(2) Process Architecture Alternatives: Figure 2.3 illustrates the following basic elements of a process architecture:

- *Data messages and control messages* – which are sent and received from one or more applications and network devices
- *Protocol tasks* – which are the units of protocol functionality that process the control messages and data messages

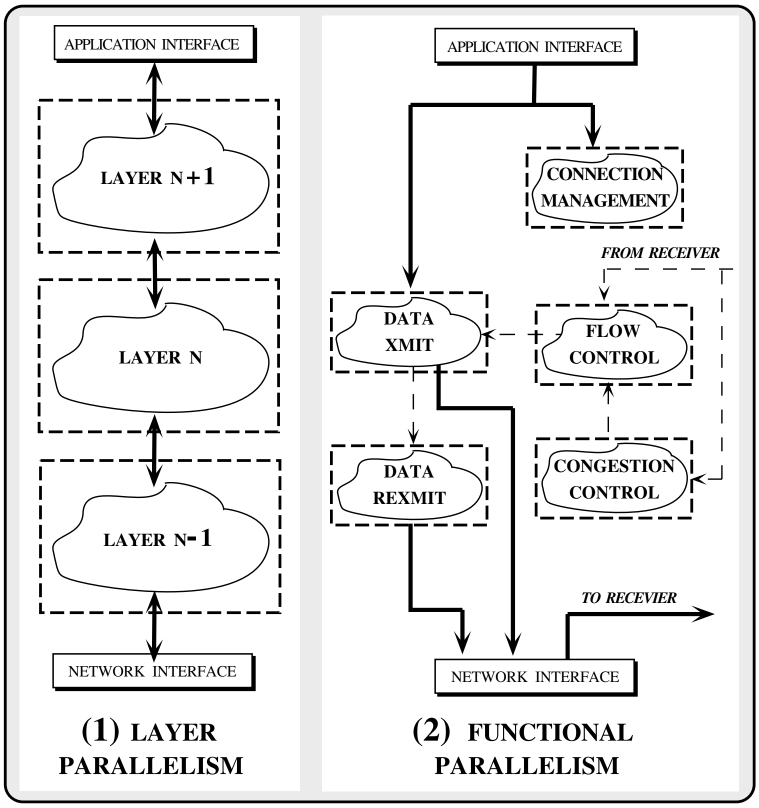

Figure 2.4: Task-based Process Architectures

- *Processing elements* (PEs) – which execute protocol tasks

There are two fundamental types of process architectures that structure these basic elements in different ways:

- *Task-based process architectures* – which bind one or more PEs to protocol processing tasks (shown in Figure 2.3 (1)). In this architecture, tasks are the active elements, whereas messages processed by the tasks are the passive elements.

- *Message-based process architectures* – which bind the PEs to the control messages and the data messages received from applications and network interfaces (shown in Figure 2.3 (2)). In this architecture, messages are the active elements, whereas tasks that process the messages are the passive elements.

In terms of functionality, protocol suites (such as the Internet and ISO OSI reference models) may be implemented using either task-based or message-based process architectures. However, each category of process architecture exhibits different structural and performance characteristics. The structural characteristics differ according to (1) the granularity of the unit(s) of protocol processing (*e.g.,* layer or function vs. connection or message) that execute in parallel, (2) the degree of CPU scalability (*i.e.,* the ability to effectively use only a fixed number of CPUs vs. a dynamically scalable amount), (3) task invocation semantics (*e.g.,* synchronous vs. asynchronous execution) and (4) the effort required to design and implement conventional and experimental protocols and services via a particular process architecture [Atk88]. In addition, different configurations of application requirements, operating system (OS) and hardware platforms, and network characteristics interact with the structural characteristics of process architectures to yield significantly different performance results. For instance, on certain general-purpose OS platforms (such as the System V STREAMS framework on multi-processor versions of UNIX), fine-grained task-based parallelism results in prohibitively high levels of synchronization overhead [SPY$^+$93]. Likewise, asynchronous, rendezvous-based task invocation semantics often result in high data movement and context switching overhead [WF93].

The remainder of this section summarizes the basic process architecture categories, classifies related work accordingly to these categories, and identifies several key factors that influence process architecture performance.

- **Task-based Process Architectures:** Task-based process architectures associate processes with clusters of one or more protocol tasks. Two common examples of task-based process architectures are *Layer Parallelism* and *Functional Parallelism.* The primary difference between these two process architectures involves the granularity of the protocol processing tasks. Protocol layers are generally more coarse-grained than protocol tasks since they cluster multiple tasks together to form a composite service (such as the end-to-end transport service provided by the ISO OSI transport layer).

- *Layer Parallelism* – Layer Parallelism is a relatively coarse-grained task-based process architecture that associates a separate process with each layer (*e.g.,* the presentation, transport, and network layers) in a protocol stack. Certain protocol header and data fields in outgoing and incoming messages may be processed in parallel as they flow through the "layer pipeline" (shown in Figure 2.4 (1)). Intra-layer buffering, inter-layer flow control, and stage balancing are generally necessary since processing activities in each layer may execute at different rates. In general, strict adherence to the layer boundaries specified by conventional communication models (such as the ISO OSI reference model) complicates stage balancing.

  An empirical study of the performance characteristics of several software architectures for implementing Layer Parallelism is presented in [WF93]. Likewise, the XINU TCP/IP implementation [CS91] uses a variant of this approach to simplify the design and implementation of its communication subsystem.

- *Functional Parallelism* – Functional Parallelism is a more fine-grained task-based process architecture that applies one or more processes to execute protocol functions (such as header composition, acknowledgement, retransmission, segmentation, reassembly, and routing) in parallel. Figure 2.4 (2) illustrates a typical Functional Parallelism design [BZ93], where protocol functions are encapsulated within parallel finite-state machines that communicate by passing control and data messages to each other. Functional Parallelism is often associated with "delayered" communication models [Haa91, Zit91, PS93] that simplify stage balancing by relaxing conventional layering boundaries in order to minimize queueing delays and "pipeline stalls" within a protocol stack.

Implementing pipelined, task-based process architectures is relatively straightforward since they typically map onto conventional layered communication models using well-structured “producer/consumer” designs [Atk88]. Moreover, minimal concurrency control mechanisms are necessary *within* a layer or function since multi-processing is typically serialized at a service access point (such as the transport or application layer interface).

• **Message-based Process Architectures:** Message-based process architectures associate processes with messages, rather than with protocol layers or protocol tasks. Two common examples of message-based process architectures are *Connectional Parallelism* and *Message Parallelism.* The primary difference between these process architectures involves the point at which messages are demultiplexed onto a process. Connectional Parallelism demultiplexes all messages bound for the same connection onto the same process, whereas Message Parallelism demultiplexes messages onto any available process.

- *Connectional Parallelism* – Connectional Parallelism is a relatively coarse-grained message-based process architecture that associates a separate process with every open connection. Figure 2.5 (1) illustrates this approach, where connections C_1, C_2, C_3, and C_4 execute in separate processes that perform the requisite protocol functions on all messages associated with their connection. Within a connection, multiple protocol processing functions are invoked serially on each message as it flows through a protocol stack. Outgoing messages typically borrow the thread of control from the application process and use it to shepard one or more messages down a protocol stack [Gar90]. For incoming messages, a device driver or packet filter [MJ93] typically performs demultiplexing operations to determine the correct process for each message. In general, Connectional Parallelism is well-suited for protocols that demultiplex early in their protocol stack

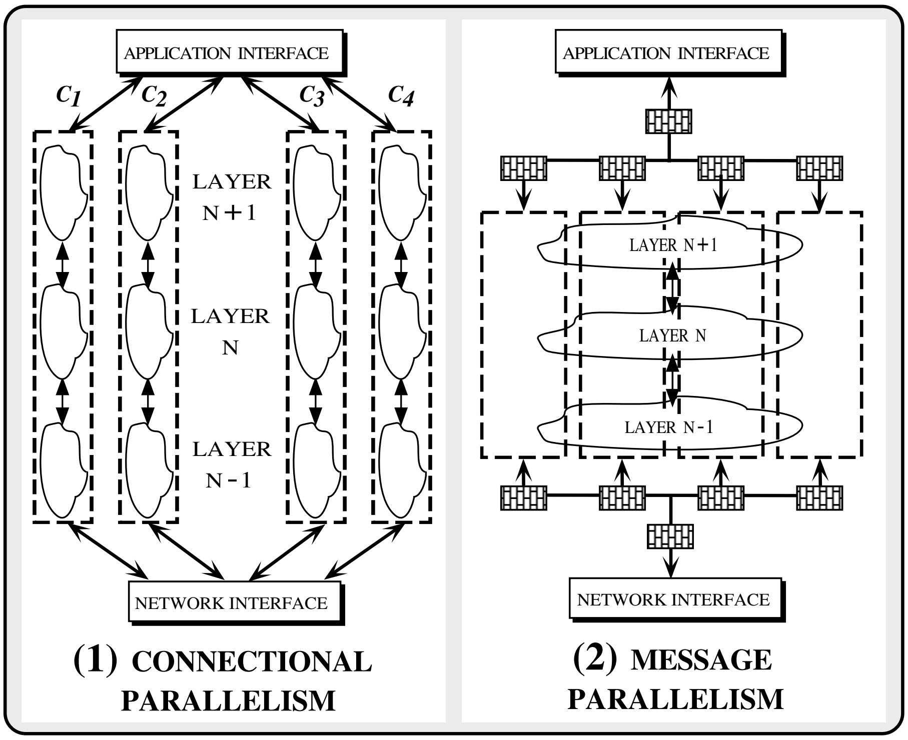

Figure 2.5: Message-based Process Architectures

since it is difficult to maintain a strict process-per-connection association across demultiplexing boundaries [Fel90].

Connectional Parallelism is relatively simple to implement if an OS allows multiple independent system calls, device interrupts, and daemon processes to operate in parallel [Gar90]. Moreover, if the number of CPUs is greater than or equal to the number of active connections, Connectional Parallelism also exhibits low communication, synchronization, and process management overhead [SPY+93] since all connection context information is localized within a particular process address space. This localization is beneficial since (1) pointers to messages may be passed between protocol layers via simple procedure calls (rather than using more complicated and costly interprocess communication mechanisms) and (2) cache affinity properties may be preserved since messages are processed largely within a single CPU cache. The primary limitation of Connectional Parallelism

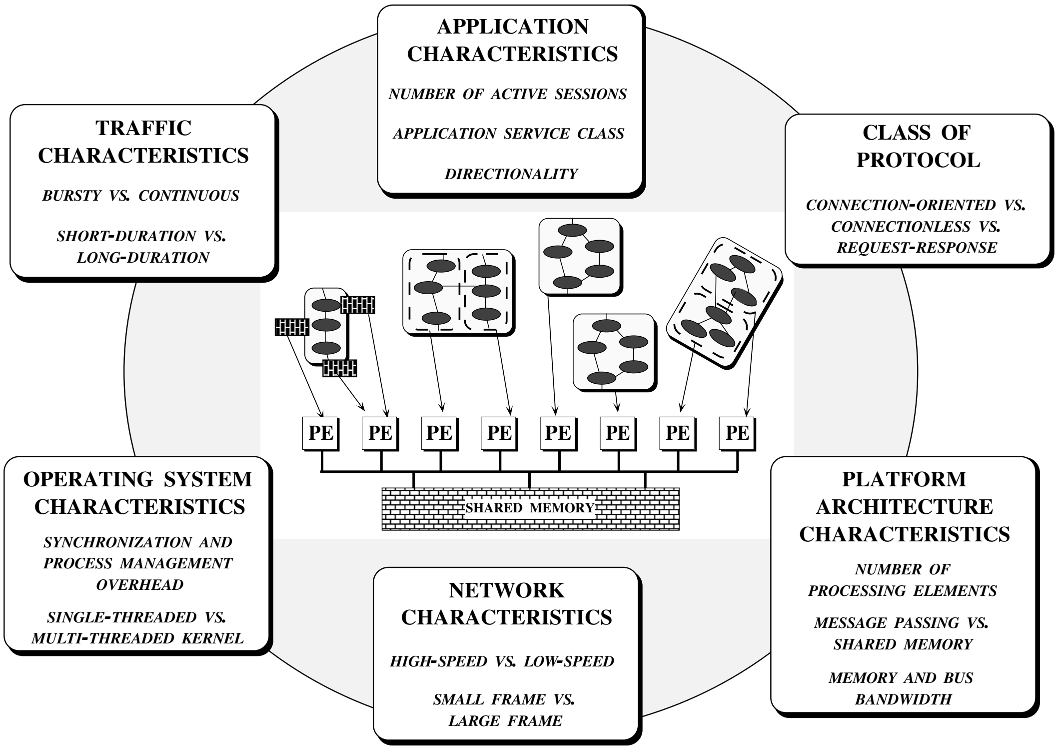

Figure 2.6: External and Internal Factors Influencing Process Architecture Performance

is that it only utilizes multi-processing to improve *aggregate* end-system performance since each individual connection still executes sequentially.

- *Message Parallelism* – Message Parallelism is a fine-grained message-based process architecture that associates a separate process with every incoming or outgoing message. As illustrated in Figure 2.5 (2), a process receives a message from an application or network interface and performs most or all of the protocol processing functions on that message. As with Connectional Parallelism, outgoing messages typically borrow the thread of control from the application that initiated the message transfer. A number of projects have discussed, simulated, or utilized Message Parallelism as the basis for their process architecture [JSB90, GNI92, HP91, Mat93, Pre93].

• **Process Architecture Performance Factors:** The performance of the process architectures described above is influenced by various *external* and *internal* factors (shown in Figure 2.6). External factors include (1) *application characteristics – e.g.,* the number of simultaneously active connections, the class of service required by applications (such as reliable/non-reliable and real-time/non-real-time), the direction of data flow (*i.e.,* uni-directional vs. bi-directional), and the type of traffic generated by applications (*e.g.,* bursty vs. continuous), (2) *protocol characteristics – e.g.,* the class of protocol (such as connectionless, connection-oriented, and request/response) used to implement application and communication subsystem services, and (3) *network characteristics – e.g.,* attributes of the underlying network environment (such as the delivery of mis-ordered data due to multipath routing [FM92]). Internal factors, on the other hand, represent hardware- and software-dependent communication subsystem implementation characteristics such as:

- *Process Management Overhead* – Process architectures exhibit different context switching and scheduling costs related to (1) the type of scheduling policies employed (*e.g.,* preemptive vs. non-preemptive), (2) the protection domain (*e.g.,* user-mode vs. kernel-mode) in which tasks within a protocol stack execute, and (3) the number of available CPUs. In general, a context switch is triggered when (1) one or more processes must sleep awaiting certain resources (such as memory buffers or I/O devices) to be come available, (2) preemptive scheduling is used and a higher priority process becomes runnable, or (3) when a currently executing process exceeds its time slice. Depending on the underlying OS and hardware platform, a context switch may be relatively time consuming due to the flushing of register windows, instruction and data caches, instruction pipelines, and translation look-aside buffers [MB91].

- *Synchronization Overhead* – Implementing communication protocols that execute correctly on multi-processor platforms requires synchronization mechanisms that serialize access to shared objects such as messages, message queues, protocol context records, and demultiplexing tables. Certain protocol and process architecture combinations (such as implementing connection-oriented protocols via Message Parallelism) may incur significant synchronization overhead from managing locks associated with these shared objects [Mat93]. In addition to reducing overall throughput, synchronization bottlenecks resulting from lock contention lead to unpredictable response times that complicate the delivery of constrained-latency applications. Other sources of synchronization overhead involve contention for shared hardware resources such as I/O buses and global memory [DAPP93]. In general, hardware contention represents an upper limit on the benefits that may accrue from multi-processing [WF93].

- *Communication Overhead* – Task-based process architectures generally require some form of interprocess communication to exchange messages between protocol processing components executing on separate CPUs. Communication costs are incurred by memory-to-memory copying, message manipulation operations (such as checksum calculations and compression), and general message passing overhead resulting from synchronization and process management operations. Common techniques for minimizing communication overhead involve (1) buffer management schemes that minimize data copying [HMPT89] and attempt to preserve cache affinity properties when exchanging messages between CPUs with separate instruction and data caches, (2) integrated layer processing techniques [CT90], and (3) single-copy network/host interface adapters [WBC+93].

- *Load Balancing* – Certain process architectures (such as Message Parallelism) have the potential for utilizing multiple CPUs equitably, whereas others (such as

Connectional, Layer, and Functional Parallelism) may under- or over-utilize the available CPUs under certain circumstances (such as bursty network and application traffic patterns or improper stage balancing).

2.3.1.2 The Virtual Memory (VM) Remapping Dimension

Regardless of the process architecture, minimizing the amount of memory-to-memory copying in a communication subsystem is essential to achieve high performance [WM87]. In general, data copying costs provide an upper bound on application throughput [CT90]. As described in Section 2.3.2.1 below, selecting an efficient message management mechanism is one method for reducing data copying overhead. A related approach described in this section uses virtual memory optimizations to avoid copying data altogether. For example, in situations where data must be transferred from one address space to another, the kernel architecture may remap the virtual memory pages by marking their page table entries as being "copy-on-write." Copy-on-write schemes physically copy memory only if a sender or receiver changes a page's contents.

Page remapping techniques are particularly useful for transferring large quantities of data between separate address spaces on the same host machine. An operation that benefits from this technique involves data transfer between user-space and kernel-space at the application interface. Rather than physically copying data from application buffers to kernel buffers, the OS may remap application pages into kernel-space instead.

Page remapping schemes are often difficult to implement efficiently in the context of communication protocols, however. For example, most remapping schemes require the alignment of data in contiguous buffers that begin on page boundaries. These alignment constraints are complicated by protocol operations that significantly enlarge or shrink the size of messages. This operations include message de-encapsulation (*i.e.,* stripping headers and trailers as messages ascend through a protocol graph), presentation layer expansion [CT90] (*e.g.,* uncompressing or decrypting an incoming message), and variable-size header options (such as those proposed to handle TCP window scaling for long-delay paths [JBB92]). Moreover, remapping may not be useful if the sender or receiver writes on the page immediately since a separate copy must be generated anyway [LMKQ89]. In addition, for small messages, more overhead may be incurred by remapping and adjusting page table entries, compared with simply copying the data in the first place.

2.3.1.3 The Event Management Dimension

Event management mechanisms provided by the kernel architecture support time-related services for user applications and other mechanisms in a communication subsystem. In general, three basic operations are exported by an event manager:

1. Registering subroutines (called "event handlers") that will be executed at some user-specified time in the future
2. Canceling a previously registered event handler
3. Invoking an event handler when its expiration time occurs

The data structures and algorithms that implement an event manager must be selected carefully so that all three types of operations are performed efficiently. In addition, the variance among different event handler invocation times should be minimized. Reducing variance is important for constrained latency applications, as well as for communication subsystems that register and execute a large number of event handlers during a given time period.

At the session architecture level, protocol implementations may use an event manager to perform certain time-related activities on network connections. In this case, a reliable connection-oriented protocol implementation registers a "retransmission-handler" with the event manager when a protocol segment is sent. The expiration time for this event is usually based on a time interval calculated from the round-trip packet estimate for that connection. If the timer expires, the event manager invokes the handler to retransmit the segment. The retransmission event handler will be canceled if an acknowledgement for the segment arrives before the timer expires.

Mechanisms for implementing event managers include *delta lists* [CS91], *timing wheels* [VL87], and heap-based [BL88] and list-based [LMKQ89] *callout queues*. These mechanisms are built atop a hardware clock mechanism. On each "clock-tick" the event manager checks whether it is time to execute any of its registered events. If one or more events must be run, the event manager invokes the associated event handler. The different event manager mechanisms may be distinguished by the following two dimensions:

(1) Search Structure: Several search structures are commonly used to implement different event management mechanisms. One approach is to sort the events by their time-to-execute value and store them in an array. A variant on this approach (used by *delta lists* and list-based *callout queues*) replaces the array with a sorted linked list to reduce the overhead of adding or deleting an event [CS91]. Another approach is to use a heap-based priority queue [BL88] instead of a sorted list or array. In this case, the average- and worst-case time complexity for inserting or deleting an entry is reduced from $O(n)$ to $O(\lg n)$. In addition to improving average-case performance, heaps also reduce the variance of event manager operations.

(2) Time Relationships: Another aspect of event management involves the "time relationships," (*i.e., absolute* vs. *relative* time) that are used to represent an event's execution time. Absolute time is generally computed in terms of a value returned by the underlying hardware clock. Heap-based search structures typically use absolute time due to the comparison properties necessary to maintain a heap as a partially-ordered, almost-complete binary tree. In contrast, relative-time may be computed as an offset from a particular starting point and is often used for a sorted linked list implementation. For example, if each item's time is stored as a *delta* relative to the previous item, the event manager need only examine the first element on every clock-tick to determine if it should execute the next registered event handler.

2.3.2 Protocol Family Architecture Dimensions

Protocol family architecture mechanisms pertain primarily to network protocols and distributed applications. In contrast, kernel architecture mechanisms are also utilized by many other applications and OS subsystems. The protocol family architecture provides intra-protocol and inter-protocol mechanisms that may be reused by protocols in many protocol families. Intra-protocol mechanisms involve the creation and deletion of sessions, whereas inter-protocol mechanisms involve message management, multiplexing and demultiplexing of messages, and layer-to-layer flow control. This section examines the inter-protocol mechanisms.

2.3.2.1 The Message Management Dimension

Communication subsystems provide mechanisms for exchanging data and control messages between communicating entities on local and remote end systems. Standard message management operations include (1) storing messages in buffers as they are received from network adapters, (2) adding and/or removing headers and trailers from messages as they pass through a protocol graph, (3) fragmenting and reassembling messages to fit into network maximum transmission units, (4) storing messages in buffers for transmission or retransmission, and (5) reordering messages received out-of-sequence [JSB90]. To improve efficiency, these operations must minimize the overhead of dynamic memory management and also avoid unnecessary data copying, as described in the following paragraphs:

(1) Dynamic Memory Management: Traditional data network traffic exhibits a bimodal distribution of sizes, ranging from large messages for bulk data transfer to small messages for remote terminal access [CDJM91]. Therefore, message managers must be capable of dynamically allocating, deallocating, and coalescing fixed-sized and variable-sized blocks of memory efficiently. However, message management schemes are often tuned for a particular range of message sizes. For instance, the BSD UNIX message management facility divides its buffers into 112 byte and 1,024 byte blocks. This leads to non-uniform performance behavior when incoming and outgoing messages vary in size between small and large blocks. As discussed in [HP91], more uniform performance is possible if message managers support a wide range of message sizes as efficiently as they support large and/or small messages.

(2) Memory Copy Avoidance: As mentioned in Section 2.3.1.2, memory-to-memory copying is a significant source of communication subsystem overhead. Naive message managers that physically copy messages between each protocol layer are prohibitively expensive. Therefore, more sophisticated implementations avoid or minimize memory-to-memory copying via techniques such as *buffer-cut-through* [WM89, ZS90] and *lazy-evaluation* [HMPT89]. Buffer-cut-through passes messages "by reference" through multiple protocol layers to reduce copying. Likewise, lazy-evaluation techniques use reference counting and buffer-sharing to minimize unnecessary copying. These schemes may be combined with the virtual memory remapping optimizations described in Section 2.3.1.2.

Message managers use different methods to reduce data copying and facilitate buffer sharing. For instance, BSD and System V UNIX attach multiple buffers together to form linked-lists of message segments. Adding data to the front or rear of a buffer list does not require any data copying since it only relinks pointers. An alternative approach uses a *directed-acyclic-graph* (DAG)-based data structure [HMPT89]. A DAG allows multiple "parents" to share all or part of a message stored in a single "child." Therefore, this method improves data sharing *between* layers in a highly-layered protocol graph. This is important for reliable protocols (such as RPC or TCP) that maintain "logical" copies of messages at certain protocol layers in case retransmission is necessary.

2.3.2.2 The Multiplexing and Demultiplexing Dimension

Multiplexing (muxing) and demultiplexing (demuxing) mechanisms select which of the sessions in an adjacent protocol layer will receive an incoming or outgoing message. A sender typically performs multiplexing, which directs outgoing messages emanating from some number of higher-layer sessions onto a smaller number of lower-layer sessions [Ten89]. Conversely, a receiver performs demultiplexing, which directs incoming messages up to their associated sessions. Multiplexing and demultiplexing are orthogonal to data copying; depending on the message management scheme, messages need not be copied as they are multiplexed and demultiplexed throughout a protocol graph [HMPT89].

Since senders generally possess knowledge of their entire transfer context (such as message destination address(es) like connection identifiers, port numbers, and/or Internet IP addresses [CT90], as well as which network interfaces to use) multiplexing may be less costly than demultiplexing. In contrast, when a network adapter receives an incoming message it generally has no prior knowledge of the message's validity or eventual destination. To obtain this information, a receiver must inspect the message header and perform demultiplexing operations that select which higher-layer protocol session(s) should receive the message.

Multiplexing and demultiplexing may be performed several times as messages move to and from network adapters, protocol layers, and user applications. Depending on the process architecture selected for a communication subsystem, multiplexing and demultiplexing activities may incur high synchronization and context switching overhead since one or more processes may need to be awakened, scheduled, and executed.

As described below, four key multiplexing and demultiplexing dimensions include *synchronization*, *layering*, *searching*, and *caching*:

(1) Synchronization: Multiplexing and demultiplexing may occur either synchronously or asynchronously, depending primarily on whether the communication subsystem uses a task-based or message-based process architecture. For example, message-based process architectures (such as the *x*-kernel) typically use synchronous multiplexing and demultiplexing since messages do not pass between separate process address spaces. Therefore, *intra-process* upcalls and subroutine calls are used to transfer messages up and down a protocol graph rather than more expensive asynchronous *inter-process* communication techniques such as message queues.

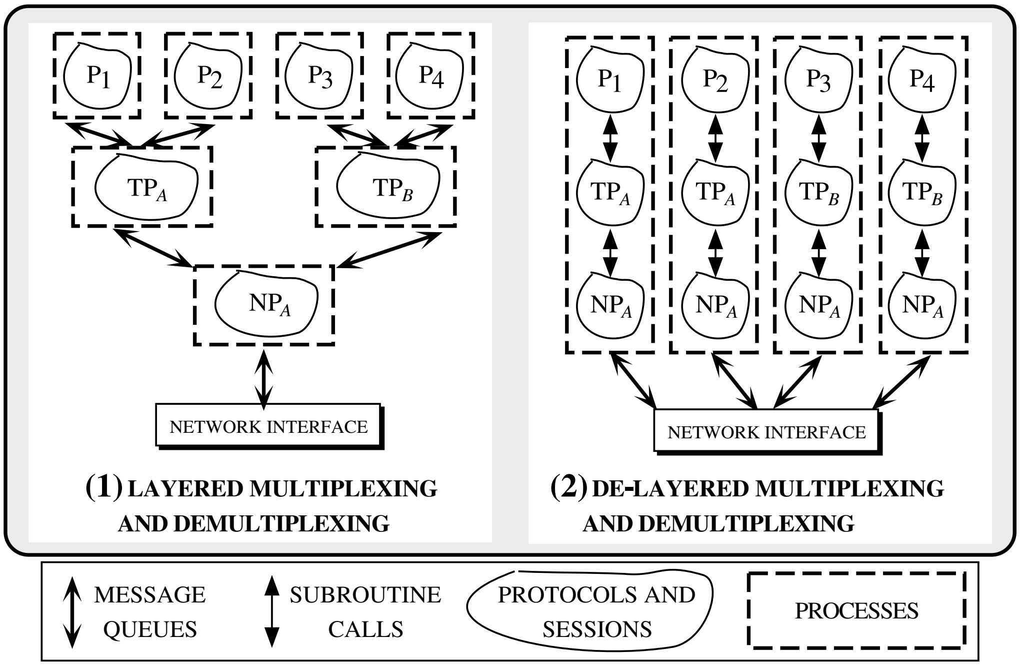

Figure 2.7: Layered and De-layered Multiplexing and Demultiplexing

In contrast, task-based process architectures (such as F-CSS [ZST93]) utilize asynchronous multiplexing and demultiplexing. In this scheme, message queues are used to buffer data passed between processes that implement a layered protocol graph. Since message queues do not necessarily block the sender, it is possible to concurrently process messages in each protocol layer, which potentially increases throughput. However, this advantage may be offset by the additional context switching and data movement overhead incurred to move messages between separate CPUs [Sch94b].

(2) Layering: As shown in Figure 2.7 (1), multiplexing and demultiplexing may occur multiple times as messages traverse up or down a protocol graph. This *layered* approach differs from the *de-layered* approach shown in Figure 2.7 (2). In the de-layered approach, multiplexing and/or demultiplexing is performed only once, usually at either the highest- or lowest-layer of a protocol graph.

The use of layered multiplexing and demultiplexing provides several benefits [Ten89]. First, it promotes modularity, since the interconnected layer components interoperate only at well-defined "service access points" (SAPs). This enables mechanisms offered at one layer to be developed independently from other layers. Second, it conserves lower-layer resources like active virtual circuits by sharing them among higher-layer sessions. Such sharing may be useful for high-volume, wide-area, leased-line communication links where it is expensive to reestablish a dedicated virtual circuit for each transmitted message. Finally, layered multiplexing and demultiplexing may be useful for coordinating related streams in multimedia applications (such as interactive teleconferencing) since messages synchronize at each SAP boundary.

The primary disadvantages of layered multiplexing and demultiplexing arise from the additional processing incurred at each layer. For example, in a task-based process architecture, multiple levels of demultiplexing may increase context switching and synchronization overhead. This overhead also enlarges packet latency variance (known as "jitter"), which is detrimental to the quality-of-service for delay- and jitter-sensitive multimedia applications such as interactive voice or video.

De-layered multiplexing and demultiplexing generally decreases jitter since there is less contention for communication subsystem resources at a single lower-layer SAP from multiple higher-layer data streams [Ten89]. However, the amount of context information stored within every intermediate protocol layer increases since sessions are not shared [Fel90]. In addition, de-layering expands the degree of demultiplexing at the lowest layer. This violates protocol layering characteristics found in conventional communication models (such as the ISO OSI reference model) since the lowest layer is now responsible for demultiplexing on addresses (such as connection identifiers or port numbers) that are actually associated with protocols several layers above in a protocol graph. Packet filters [MJ93] are a technique used to address this issue. Packet filters allow applications and higher-level protocols to "program" a network interface so that particular types of incoming PDUs are demultiplexed directly to them, rather than passing through a series of intervening protocol layers first.

Note that the use of de-layered multiplexing and demultiplexing interacts with the choice of process architecture. For example, Connectional Parallelism is enhanced by protocols that demultiplex early in their protocol stack since it is difficult to maintain a strict process-per-connection association across demultiplexing boundaries [Fel90].

(3) Searching: Some type of search algorithm is required to implement multiplexing and demultiplexing schemes. Several common search algorithms include *direct indexing*, *sequential search*, and *hashing*. Each algorithm uses an *external identifier* search key (such as a network address, port number, or type-of-service field) to locate an *internal identifier* (such as a pointer to a protocol control block or a network interface) that specifies the appropriate session context record.

Transport protocols such as TP4 and VMTP pre-compute *connection identifiers* during connection establishment to simplify subsequent demultiplexing operations. If these identifiers have a small range of values, a demultiplexing operation may simply index directly into an array-based search structure to locate the associated session context record. Alternatively, a sequential search may be used if a protocol does not support connection identifiers, or if the range of identifier values is large and sparse. For example, BSD UNIX demultiplexes TCP and UDP associations by performing a sequential search on external identifiers represented by a <*source addr, source port, destination port*> tuple. Although sequential search is simple to implement, it does not scale up well if the communication subsystem has hundreds or thousands of external identifiers representing active connections. In this case, a more efficient search algorithm (such as bucket-chained hashing) may be required.

(4) Caching: Several additional optimizations may be used to augment the search algorithms discussed above. These optimizations include (1) single- or multiple-item caches and (2) list reorganization heuristics that move recently accessed control blocks to the front of the search list or hash bucket-chain. A single-item cache is relatively efficient if the arrival and departure of application data exhibit "message-train" behavior. A message-train is a sequence of back-to-back messages that are all destined for the same higher-level session. However, single-item caching is insufficient if application traffic behavior is less uniform [MD91]. When calculating how well a particular caching scheme affects the cost of demultiplexing it is important to consider (1) the *miss ratio*, which represents how many times the desired external identifier is *not* in the cache and (2) the number of list entries that must be examined when a cache miss occurs. In general, the longer the search list, the higher the cost of a cache miss.

The choice of search algorithm and caching optimization impacts overall communication subsystem and protocol performance significantly. When combined with caching, hashing produces a measurable improvement for searching large lists of control blocks that correspond to active network connections [HP91].

2.3.2.3 The Layer-to-Layer Flow Control Dimension

Layer-to-layer flow control regulates the rate of speed and amount of data that is processed at various levels in a communication subsystem. For example, flow control is performed at the application interface by suspending user processes that attempt to send and/or receive more data than end-to-end session buffers are capable of handling. Likewise, within the protocol family architecture level, layer-to-layer flow control prevents higher-layer protocol components from flooding lower-layers with more messages than they are equipped to process and/or buffer.

Layer-to-layer flow control has a significant impact on protocol performance. For instance, empirical studies [CWWS92] demonstrate the importance of matching buffer sizes and flow control strategies at each layer in the protocol family architecture. Inefficiencies may result if buffer sizes are not matched appropriately in adjacent layers, thereby causing excessive segmentation/reassembly and additional transmission delays.

Two general mechanisms for controlling the layer-to-layer flow of messages include the *per-queue* flow control and *per-process* flow control schemes outlined below:

- **Per-Queue Flow Control:** Flow control may be implemented by enforcing a limit on the number of messages or total number of bytes that are queued between sessions in adjacent protocol layers. For example, a task-based process architecture may limit the size of the message queues that store information passed between adjacent sessions and/or user processes. This approach has the advantage that it enables control of resource utilization at a fairly fine-grain level (such as per-connection).

- **Per-Process Flow Control:** Flow control may also be performed in a more coarse-grained manner at the per-process level. This approach is typically used by message-based process architectures. For example, in the *x*-kernel, an incoming message is discarded at a network interface if a light-weight process is not available to shepard an incoming message up through a protocol graph. The advantage of this approach is that it reduces queueing complexity at higher-layers. However, it may unfairly penalize connections that are not responsible for causing message congestion on an end system.

2.4 Survey of Existing OS Communication Subsystem Architectures

A number of framework have emerged to simplify the development and configuration of communication subsystems by inter-connecting session and protocol family architecture components. In general, these frameworks encourage the development of standard communication-related components (such as message managers, timer-based event dispatchers, and connection demultiplexors [HMPT89], and various protocol functions [SSS+93]) by decoupling protocol processing functionality from the surrounding framework infrastructure. This section surveys the communication subsystem architectures for the System V UNIX, BSD UNIX, *x*-kernel, and Choices operating systems. Unless otherwise noted, the systems described include System V Release 4, BSD 4.3 Tahoe, *x*-kernel 3.2, and Choices 6.16.91. Section 2.4.1 gives a brief summary of each system. Section 2.4.2 compares and contrasts each system using the taxonomy dimensions listed in Table 2.1.

2.4.1 System Overviews

This section outlines the primary software components and process architectures for each surveyed communication subsystem in order to highlight the design decisions made by actual systems. In addition, a communication subsystem *profile* corresponding to the taxonomy depicted in Table 2.1 is presented along with each overview (note that ND stands for "not defined").

2.4.1.1 System V STREAMS

Process Architecture	(1) coroutines, (2) task-based (process-per-module)
VM Remapping	none
Event Management	(1) absolute, (2) heap
Message Buffering	(1) uniform, (2) list-based
Muxing/Demuxing	(1) asynchronous, (2) layered, (3) ND, (4) ND
Flow Control	per-queue

Table 2.2: STREAMS Profile

The System V STREAMS architecture emphasizes modular components that possess uniform interfaces. It was initially developed to support the flexible composition of terminal drivers in UNIX [Rit84]. It was later extended to support network protocols and local IPC via *multiplexor drivers* and STREAM *pipes*, respectively [Rag93]. The Table 2.2 illustrates the communication subsystem profile for System V STREAMS. In the discussion below, the uppercase term "STREAMS" refers to the overall System V communication subsystem mechanism, whereas the term "Stream" refers to a full-duplex protocol processing and data transfer path between a user application and a device driver.

As shown in Figure 2.8, the main components in the System V STREAMS architecture include STREAM *heads*, STREAM *modules*, STREAM *multiplexors*, and STREAM *drivers*. A STREAM head segments the user data into discrete messages. These messages are passed "downstream" from the STREAM head though zero or more STREAM modules and multiplexors to the STREAM driver, where they are transmitted by a network adapter to the appropriate network. Likewise, the driver also receives incoming messages from the network. These messages are passed "upstream" through the modules to the STREAM head, where a user process may retrieve them. STREAM modules and multiplexors may be inserted and/or removed dynamically between the head

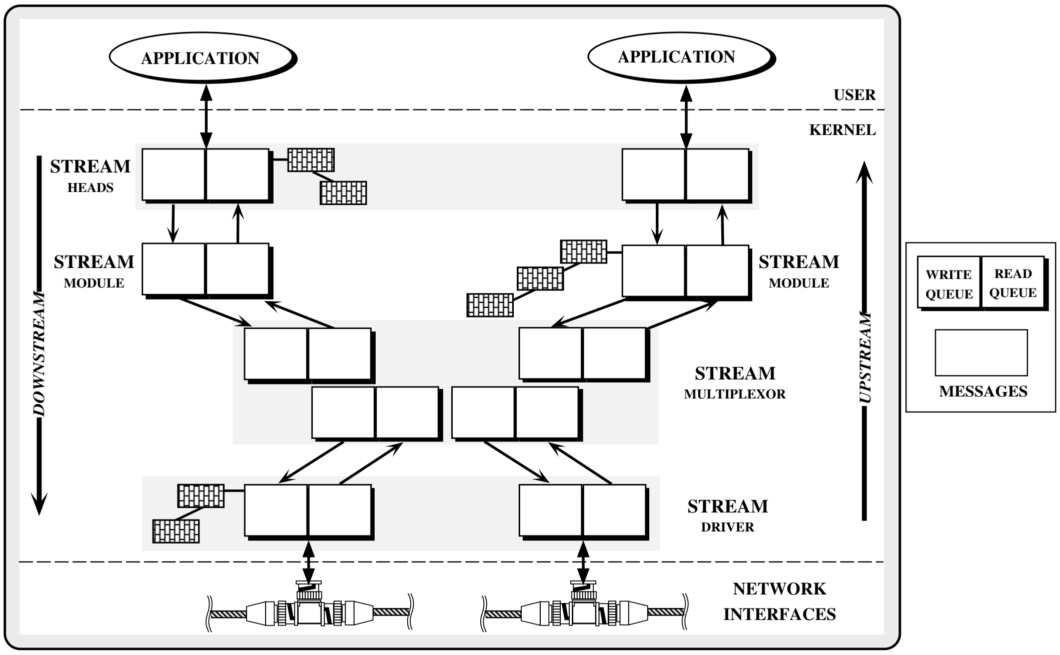

Figure 2.8: System V STREAMS Architecture

and the driver. Each module or multiplexor implements protocol processing mechanisms like encryption, compression, reliable message delivery, and routing. The following paragraphs describe each STREAMS component:

• **STREAM Heads:** STREAM heads are situated on "top" of a Stream, directly "below" the user process (as shown in Figure 2.8). STREAM heads provide a queueing point for exchanging data and control information between an application (running as a user process) and a Stream (running in the kernel). Each STREAM component is linked together with its adjacent components via a pair of queues: one for reading and the other for writing. These queues hold lists of messages sorted by up to 256 different priority levels. Since the System V application interface does not use virtual memory remapping techniques, the STREAM head also performs memory-to-memory copying to transfer data between a user process and the kernel.

• **STREAM Modules:** Each STREAM module performs its protocol processing operations on the data it receives before forwarding the data to the next module. In this way, STREAM modules are analogous to "filter" programs in a UNIX shell pipeline. Unlike a UNIX pipeline, however, data is passed as discrete messages between modules, rather than as a byte-stream. Applications may "push" and/or "pop" STREAM modules on or off a Stream dynamically in "last-in, first-out" (LIFO) order. Each read and write queue in a module contains pointers to subroutines that (1) implement the module's protocol processing operations and (2) regulate layer-to-layer message flow between modules.

Two subroutines associated with each queue are called `put` and `service`. The `put` subroutine typically performs synchronous message processing when invoked by an adjacent queue (*e.g.,* when a user process sends a message downstream or a message arrives on a network interface). It performs protocol processing operations that must be invoked immediately (such as handling high-priority TCP "urgent data" messages).

The `service` subroutine, on the other hand, is used for protocol operations that either do not execute in a short, fixed amount of time (*e.g.,* performing a three-way handshake to establish an end-to-end network connection) or that will block indefinitely (*e.g.,* due to layer-to-layer flow control). The `service` subroutines in adjacent modules generally interact in a coroutine-like manner. For example, when a queue's `service` subroutine is run, it performs protocol processing operations on all the messages waiting in the queue. When the `service` subroutine completes, the messages it processed will have been passed to the appropriate adjacent STREAM module in the Stream. Next, the `service` routine for any STREAM modules that now have new messages in their queue(s) is scheduled to run.

• **STREAM Multiplexors:** STREAM multiplexors may be linked between a STREAM head and a STREAM driver, similar to STREAM modules. Unlike a STREAM module, however, a multiplexor driver is linked with *multiple* Streams residing directly "above" or "below" it. Multiplexors are used to implement network protocols such as TCP and IP that receive data from multiple sources (*e.g.,* different user processes) and send data to multiple sources (*e.g.,* different network interfaces).

• **STREAM Drivers:** STREAM drivers are connected at the "bottom" of a Stream. They typically manage hardware devices, performing activities such as handling network adapter interrupts and converting incoming packets into messages suitable for upstream modules and multiplexors.

• **Messages:** Data is passed between STREAMS components in discrete chunks via an abstraction called a *message*. Messages consist of a *control* block and one or more *data* blocks. The control block typically contains bookkeeping information such as destination addresses and length fields). The data blocks generally contain the actual message contents, *i.e.,* its "payload."

To minimize memory-to-memory copying costs, pointers to message blocks are passed upstream and downstream. A message is represented as a <*message control block, data control block, variable length data buffer*> tuple. This tuple minimizes memory-to-memory copying costs by sharing a common <*data buffer*> among several <*message control block, data control block*> portions.

The traditional System V STREAMS communication subsystem supports a variant of the task-based process architecture known as "process-per-module" that associates a "logical" process with a STREAM module's `service` subroutine. This process-per-module approach is implemented by scheduling and executing the `service` subroutines associated with the read and write queues in a STREAM module. Originally, the `service` procedures were run only at certain times (such as just before returning from a system call and just before a user process was put to sleep). Unfortunately, this design made it difficult to support applications with isochronous or constrained latency requirements since STREAM modules were not scheduled to run with any precise real-time guarantees. In addition, these subroutines execute outside the context of any kernel or user process, thereby avoiding the standard UNIX kernel process scheduling mechanism. This design represents an effort to (1) minimize the kernel state information required for process management and (2) reduce context switching overhead when moving messages between module queues.

Many STREAMS implementations [EKB+92, Gar90, Pre93, SPY+93] utilize shared memory, symmetric multi-processing capabilities within a multi-threaded kernel address space. These implementations supports various levels of STREAMS concurrency. These concurrency levels range from relatively fine-grain parallelism (such as *queue-level* with one light-weight process (LWP) for the STREAM module read queue and one LWP for the STREAM module write queue and *queue-pair-level* with one LWP shared by a STREAM module queue pair) to more coarse-grained approaches (such as *module-level* with one LWP shared across all instances of a STREAM module and *module-class-level* with one LWP shared across a particular class of STREAM modules).

2.4.1.2 BSD UNIX Network Subsystem

BSD UNIX provides a communication subsystem framework that supports multiple protocol families such as the Internet, XNS, and OSI protocols [LMKQ89]. BSD provides a general-purpose application interface called *sockets*. Sockets are bi-directional communication channels that transfer data between unrelated processes on local and remote hosts. Table 2.3 illustrates the communication subsystem profile for BSD UNIX.

Process Architecture	(1) single-threaded, (2) hybrid message-based
VM Remapping	incoming
Event Management	(1) relative, (2) linked list
Message Buffering	(1) non-uniform, (2) list-based
Muxing/Demuxing	(1) hybrid, (2) layered, (3) sequential, (4) single-item
Flow Control	ND

Table 2.3: BSD UNIX Profile

The concept of a *communication domain* is central to BSD's multiple protocol family design. A domain specifies both a protocol family and an address family. Each protocol family implements a set of protocols corresponding to standard socket types in the domain. For example, `SOCK_STREAM` provides reliable byte-stream communication and `SOCK_DGRAM` provides unreliable datagram communication. 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 message is intended for). The standard BSD release supports address families for the Internet domain, XEROX NS domain, OSI domain, and UNIX domain (which only exchanges information between sockets in processes on a local host).

There are three main layers in the BSD communication subsystem design: the *socket layer*, *protocol layer*, and *network interface layer*. Data are exchanged between these layers in discrete chunks called *mbufs*. Socket layer mechanisms are similar to System V STREAM heads. One difference is that a STREAM head supports up to 256 levels of message priority, whereas sockets only provide 2 levels ("in-band" and "out-of-band"). The protocol layer coordinates algorithms and data structures that implement the various BSD protocol families. The network interface layer provides a software veneer for accessing the underlying network adapter hardware. The following paragraphs describe the major BSD protocol layer components in detail:

• **The Socket Layer:** A socket is a *typed* object that represents a bi-directional endpoint of communication. Sockets provide a queueing point for data that is transmitted and received between user applications running as user processes and the protocol layers running in the OS kernel. Open sockets are identified via *socket descriptors*. These descriptors index into a kernel table containing socket-related information such as send and receive buffer queues, the socket type, and pointers to the associated protocol layer. When a socket is created, a new table slot is initialized based on the specified "socket type" (*e.g.,* `SOCK_STREAM` or `SOCK_DGRAM`). Socket descriptors share the same name space as UNIX file descriptors. This allows many UNIX applications to communicate transparently using different kinds of devices such as remote network connections, files, terminals, printers, and tape drives.

• **The Protocol Layer:** BSD's protocol layer contains multiple components organized using a dispatch table format. Unlike STREAMS, the BSD network architecture does not allow arbitrary configuration of protocol components at run-time. Instead, protocol families are created by associating certain components with one another when a kernel image is statically linked.

In the Internet protocol family, the TCP component is linked above the IP component. Each protocol component stores session context information in *control blocks* that represent open end-to-end network sessions. Internet domain control blocks include the `inpcb` (which stores the source and destination host addresses and port numbers) and the `tcpcb` (which stores the TCP state machine variables such as sequence numbers, retransmission timer values, and statistics for network management). Each `inpcb` also contains links to sibling `inpcbs` (which store information on other active network sessions in the protocol layer), back-pointers to the socket data structure associated with the protocol session, and other relevant information such as routing-table entries or network interface addresses.

• **The Network Interface Layer:** Messages arriving on network interfaces are handled by a software interrupt-based mechanism, as opposed to dedicating a separate kernel "process" to perform network I/O. Interrupts are used for two primary reasons: (1) they reduce the context switching overhead that would result from using separate processes and (2) the BSD kernel is not multi-threaded. There are two levels of interrupts: SPLNET and SPLIMP. SPLNET has higher priority and is generated when a network adapter signals that a message has arrived on an interface. However, since hardware interrupts cannot be masked for very long without causing other OS devices to timeout and fail, a lower priority software interrupt level named SPLIMP actually invokes the higher-layer protocol processing.

For example, when an SPLNET hardware interrupt occurs, the incoming message is placed in the appropriate network interface protocol queue (*e.g.,* the queue associated with the IP protocol). Next, an SPLIMP software interrupt is posted, informing the kernel that higher-layer protocols should be run when the interrupt priority level falls below SPLIMP. When the SPLIMP interrupt handler is run, the message is removed from the queue and processed to completion by higher-layer protocols. If a message is not discarded by a protocol (*e.g.,* due to a checksum error) it typically ends up in a socket receive queue, where a user process may retrieve it.

• **Mbufs:** BSD UNIX uses the mbuf data structure to manage messages as they flow between levels in the network subsystem. An mbuf's representation and its associated operations are similar to the System V STREAMS message abstraction. Mbuf operations include subroutines for allocating and freeing mbufs and lists of mbufs, as well as for adding and deleting data to an mbuf list. These subroutines are designed to minimize memory-to-memory coping. Mbufs store lists of incoming messages and outgoing protocol segments, as well as other dynamically allocated objects like the socket data structure. There are two primary types of mbufs: *small mbufs*, which contain 128 bytes (112 bytes of which are used to hold actual data), and *cluster mbufs*, which use 1 kbyte pages to minimize fragmentation and reduce copying costs via reference counting.

BSD uses a single-threaded, hybrid message-based process architecture residing entirely in the kernel. User processes enter the kernel when they invoke a socket-related system call. Due to flow control, multiple user processes that are sending data to "lower" protocol layers residing in the kernel may be blocked simultaneously at the socket layer. Blocked processes are suspended from sending messages down to the network interface layer until flow control conditions abate. In contrast, since the BSD kernel is single-threaded, only one thread of control executes to process incoming messages up through the higher protocol layers.

2.4.1.3 x-kernel

The *x*-kernel is a modular, extensible communication subsystem kernel architecture designed to support prototyping and experimentation with alternative protocol and session architectures [HP91]. It was developed to demonstrate that layering and modularity are not inherently detrimental to network protocol performance [HP91]. The *x*-kernel supports protocol graphs that implement a wide range of standard and experimental protocol families, including TCP/IP, Sun RPC, Sprite RCP, VMTP, NFS, and

Process Architecture	(1) LWP, (2) message-based
VM Remapping	incoming/outgoing
Event Management	(1) relative, (2) linked list
Message Buffering	(1) uniform, (2) DAG-based
Muxing/Demuxing	(1) synchronous, (2) layered, (3) hashing, (4) single-item
Flow Control	per-process

Table 2.4: *x*-kernel Profile

Psync [PBS89]. Unlike BSD UNIX, whose protocol family architecture is characterized by a static, relatively monolithic protocol graph, the *x*-kernel supports dynamic, highly-layered protocol graphs. Table 2.4 illustrates the communication subsystem profile for the *x*-kernel.

The *x*-kernel's protocol family architecture provides highly uniform interfaces to its mechanisms, which manage three communication abstractions that comprise protocol graphs [HP91]: *protocol objects*, *session objects*, and *message objects*. These abstractions are supported by other reusable software components that include a *message manager* (an abstract data type that encapsulates messages exchanged between session and protocol objects), a *map manager* (used for demultiplexing incoming messages between adjacent protocols and sessions), and an *event manager* (based upon *timing wheels* [VL87] and used for timer-driven activities like TCP's adaptive retransmission algorithm). In addition, the *x*-kernel provides a standard library of *micro-protocols*. These are reusable, modular software components that implement mechanisms common to many protocols (such as include sliding window transmission and adaptive retransmission schemes, request-response RPC mechanisms, and a "blast" protocol that uses selective retransmission to reduce channel utilization [OP92]). The following paragraphs describe the *x*-kernel's primary software components:

• **Protocol Objects:** Protocol objects are software abstractions that represent network protocols in the *x*-kernel. Protocol objects belong to one of two "realms," either the *asynchronous* realm (*e.g.,* TCP, IP, UDP) or the *synchronous* realm (*e.g.,* RPC). The *x*-kernel implements a protocol graph by combining one or more protocol objects. A protocol object contains a standard set of subroutines that provide uniform interfaces for two major services: (1) creating and destroying session objects (which maintain a network connection's context information) and (2) demultiplexing message objects up to the appropriate higher-layer session objects. The *x*-kernel uses its map manager abstraction to implement efficient demultiplexing. The map manager associates external identifiers (*e.g.,* TCP port numbers or IP addresses) with internal data structures (*e.g.,* session control blocks). It is implemented as a chained-hashing scheme with a single-item cache.

• **Session Objects:** A session object maintains context information associated with a local end-point of a connection. For example, a session object stores the context information for an active TCP state machine. Protocol objects create and destroy session objects dynamically. When an application opens multiple connections, one or more session objects will be created within the appropriate protocol objects in a protocol graph. The *x*-kernel supports operations on session objects that involve "layer-to-layer" activities such as exchanging messages between higher-level and lower-level sessions. However, the *x*-kernel's protocol family architecture framework does not provide standard mechanisms for "end-to-end" session architecture activities such as connection management, error detection, or end-to-end flow control. A related project, Avoca, builds upon the basic *x*-kernel facilities to provide these end-to-end session services [OP92].

• **Message Objects:** Message objects encapsulate control and user data information that flows "upwards" or "downwards" through a graph of session and protocol objects.

In order to decrease memory-to-memory copying and to implement message operations efficiently, message objects are implemented using a "directed-acyclic-graph" (DAG)-based data structure. This DAG-based scheme uses "lazy-evaluation" to avoid unnecessary data copying when passing messages between protocol layers [HMPT89]. It also stores message headers in a separate "header stack" and uses pointer arithmetic on this stack to reduce the cost of prepending or stripping message headers.

The *x*-kernel employs a "process-per-message" message-based process architecture that resides in either the OS kernel or in user-space. The kernel implementation maintains a pool of light-weight processes (LWPs). When a message arrives at a network interface, a separate LWP is dispatched from the pool to shepard the message upwards through the graph of protocol and session objects. In general, only one context switch is required to shepard a message through the protocol graph, regardless of the number of intervening protocol layers. The *x*-kernel also supports other context switch optimizations that (1) allow user processes to transform into kernel processes via system calls when sending message and (2) allow kernel processes to transform into user processes via upcalls when receiving messages [Cla85].

2.4.1.4 The Conduit Framework

The Conduit framework provides the protocol family architecture, session architecture, and application interface for the Choices operating system [CRJ87]. Choices was developed to investigate the suitability of object-oriented techniques for designing and implementing OS kernel and networking mechanisms.[4] For example, the design of ZOOT (the Choices TCP/IP implementation) uses object-oriented language constructs

[4]Choices and the Conduit framework are written using C++. All the other surveyed systems are written in C.

Process Architecture	(1) LWP, (2) hybrid (process-per-buffer)
VM Remapping	none
Event Management	ND
Message Buffering	(1) uniform, (2) list-based
Muxing/Demuxing	(1) ND, (2) layered, (3) ND, (4) ND
Flow Control	ND

Table 2.5: Conduit Profile

and design methods such as inheritance, dynamic binding, and delegation [ZJ91] to implement the TCP state machine in a highly modular fashion. Together, Choices and the Conduit framework provide a general-purpose communication subsystem. Table 2.5 illustrates the communication subsystem profile for the Choices Conduit framework. In the discussion below, the term "Conduit framework" refers to the overall communication subsystem, whereas a "`Conduit`" corresponds to an abstract data type used to construct and coordinate various network protocols.

The three major components in the Conduit framework are: `Conduits`, `Conduit Messages`, and `Conduit Addresses`. A `Conduit` is a bi-directional communication abstraction, similar to a System V STREAM module. It exports operations that allow `Conduits` (1) to link together and (2) to exchange messages with adjacently linked `Conduits`. `Conduit Messages` are typed objects exchanged between adjacent `Conduits` in a protocol graph. `Conduit Addresses` are utilized by `Conduits` to determine where to deliver `Conduit Messages`. All three components are described in the following paragraphs:

• **The Conduit Base Class and Subclasses:** A `Conduit` provides the basis for implementing many types of network protocols including connectionless (*e.g.,* Ethernet, IP, ICMP, and UDP), connection-oriented (*e.g.,* TCP and TP4), and request-response (*e.g.,* RPC and NFS) protocols. It is represented as a C++ base class that provides two types of operations that are inherited and/or redefined by derived subclasses. One type of operation composes protocol graphs by connecting and disconnecting `Conduits` instances. The other type of operation inserts messages into the "top" and/or "bottom" of a `Conduit`. Each `Conduit` has two ends for processing data and control messages: the top end corresponds to messages flowing *down* from an application; the bottom end corresponds to messages flowing *up* from a network interface.

The Conduit framework uses C++ mechanisms such as inheritance and dynamic binding to express the commonality between the `Conduit` base class and its various subclasses. These subclasses represent *specializations* of abstract network protocol classes that provide *Virtual Circuit* and *Datagram* services. For instance, the `Virtual_Circuit_Conduit` and `Datagram_Conduit` are standard Conduit framework subclasses. Both subclasses export the "connect, disconnect, and message insertion" mechanisms inherited from the `Conduit` base class. In addition, they also extend the base class interface by supplying operations that implement their particular mechanisms. A `Virtual_Circuit_Conduit` provides an interface for managing end-to-end "sliding window" flow control. It also specifies other properities associated with virtual circuit protocols such as reliable, in-order, unduplicated data delivery. These two subclasses are themselves used as base classes for further specializations such as the `TCP_Conduit` and `Ethernet_Conduit` subclasses, respectively.

• **Conduit Messages:** All messages that flow between `Conduits` have a particular type. This type indicates the contents of a message (*e.g.,* its header and data format) and specifies the operations that may be performed on the message. Messages are derived from a C++ base class that provides the foundation for subsequent inherited subclasses. Different message subclasses are associated with the different `Conduit` subclasses that represent different network protocols. For example, the `IP_Message` and `TCP_Message` subclasses correspond to the `IP_Conduits` and `TCP_Conduits`, respectively. `Conduit Message` subclasses may also encapsulate other messages. For instance, an IP message may contain a TCP, UDP, or ICMP message in its data portion.

• **Conduit Addresses:** `Conduit Addresses` indicate where to deliver `Conduit Messages`. The three main types of `Conduit Addresses` are *explicit*, *implicit*, and *embedded*. Explicit addresses identify entities that have a "well-known" format (such as IP addresses). Implicit addresses, on the other hand, are "keys" that identify particular session control blocks associated with active network connections. For example, a socket descriptor in BSD UNIX is an implicit address that references a session control block. Finally, an embedded address is an explicit address that forms part of a message header. For example, the fixed-length, 14 byte Ethernet headers are represented as embedded addresses since passing a separate explicit address object is neither time nor space efficient.

The Conduit framework is implemented in user-space and the relationship of processes to `Conduits` and `Conduit Messages` is a hybrid between message-based and task-based process architectures. Messages are escorted through the Conduit framework protocol graph via "walker-processes," which are similar to the *x*-kernel "process-per-message" mechanism. Depending on certain conditions, a walker process escorts outgoing messages most of the way up or down a protocol graph. However, when a message crosses an address space boundary or must be stored in a buffer due to flow control, it remains there until it is moved to an adjacent `Conduit`. This movement may result from either (1) a daemon process residing in the `Conduit` that buffered the message or (2) another process that knows how to retrieve the message from the flow control buffer. In general, the number of processes required to escort a message through the chain of `Conduits` corresponds to the number of flow control buffers between the application and network interface layer.

2.4.2 Communication Subsystem Comparisons

This section compares and contrasts the four surveyed communication subsystems using the taxonomy dimensions and alternatives presented in Table 2.1. Section 2.4.2.1 focuses on the kernel architecture dimensions described in Section 2.3.1 and Section 2.4.2.2 focuses on the protocol family architecture dimensions described in Section 2.3.2.

2.4.2.1 Comparison of Kernel Architecture Dimensions

• **The Process Architecture Dimension:** The surveyed communication subsystems exhibit a range of process architectures. The conventional System V STREAMS implementation uses a variant of the task-based process architecture known as a "process-per-module" approach. However, as described in Section 2.4.1.1, the standard System V STREAMS approach does not associate a heavy-weight OS process per module in an effort to reduce context switching overhead and minimize kernel state information required for process management.

The *x*-kernel and BSD UNIX utilize variants of a message-based process architecture. The *x*-kernel supports highly-layered protocol graphs that use a "process-per-message" approach that is tuned to avoid excessive context switching and IPC overhead. BSD UNIX uses a message-based approach that behaves differently depending on whether messages are flowing "up" or "down" through a protocol graph. For example, BSD allows multiple processes into the kernel for outgoing messages, but permits only one process to handle incoming messages.

The Conduit framework uses a "process-per-buffer" approach, which is a hybrid between "process-per-message" and "process-per-module." Each `Conduit` containing a flow control buffer may be associated with a separate light-weight process.

• **The Virtual Memory Remapping Dimension:** Recent versions of *x*-kernel provide virtual memory remapping [HMPT89] for transferring messages between application process and the kernel. The Conduit framework, System V STREAMS and BSD UNIX, on the other hand, do not generally provide this support.

• **The Event Management Dimension:** BSD UNIX stores pointers to subroutines in a *linked-list callout queue*. These preregistered subroutines are called when a timer expires. System V, on the other hand, maintains a *heap-based callout table*, rather than a sorted list or array. The heap-based implementation outperforms the linked-list approach under heavy loads [BL88]. The *x*-kernel uses *timing wheels* [VL87] instead of callout lists or heaps.

2.4.2.2 Comparison of Protocol Family Architecture Dimensions

Compared with the other surveyed communication subsystems, the *x*-kernel is generally more comprehensive in supplying the interfaces and mechanisms for its protocol family architecture components. For example, it provides uniform interfaces for operations that manage the protocol, session, and message objects comprising its highly-layered protocol graphs. In addition, it also specifies mechanisms for event management and multiplexing and demultiplexing activities. System V STREAMS specifies interfaces for the primary STREAM components, along with certain operations involving layer-to-layer flow control. BSD UNIX and the Conduit framework, on the other hand, do not systematically specify the session, demultiplexing, and flow control mechanisms in their protocol family architecture.

• **The Message Management Dimension:** Both System V STREAMS messages and BSD mbufs use a linear-list-based approach. In contrast, the *x*-kernel uses a DAG-based approach that separates messages into "header stacks" and "data graphs." The *x*-kernel uses this more complex DAG-based message manager to handle certain requirements of highly-layered protocol graphs (such as minimizing the amount of memory-to-memory copying between protocol layers).

• **The Multiplexing and Demultiplexing Dimension:** The four surveyed communication subsystems possess a wide range of multiplexing and demultiplexing strategies. The *x*-kernel provides the most systematic support for these operations. It provides a *map manager* that uses a hash table mechanism with a single-item cache. The other communication subsystems provide less systematic and non-uniform mechanisms.

In particular, System V STREAMS and the Conduit framework do not define a standard multiplexing and demultiplexing interface. Moreover, for outgoing messages, the Conduit framework involves an extra multiplexing operation compared to the *x*-kernel scheme. In the *x*-kernel, a single operation transfers outgoing messages from a higher-layer session object down to lower-layer session object. A `Conduit`, on the other hand, requires two operations to send a message: (1) it locates the appropriate session connection descriptor associated with the lower-level `Conduit` and (2) then passes the message down to that associated `Conduit`.

The BSD UNIX multiplexing and demultiplexing mechanisms differ depending on which protocol component and protocol family are involved. For instance, its IP implementation uses the 8-bit IP message type-of-service field to index into an array containing 256 entries that correspond to higher-layer protocol control structures. On the other hand, its TCP implementation uses sequential search with a one-item cache to demultiplex incoming messages to the appropriate connection session. As described in Section 2.3.2.2, this implementation is inefficient when application data arrival patterns do not form message-trains [MD91].

• **The Layer-to-Layer Flow Control Dimension:** With the exception of System V STREAMS, the surveyed communication subsystems do not provide uniform layer-to-layer flow control mechanisms. Each STREAM module contains high- and low-watermarks that manage flow control between adjacent modules. Downstream flow control operates from the "bottom up." If all STREAM modules on a Stream cooperate, it is possible to control the amount and the rate of messages by exerting "back-pressure" up a stack of STREAM modules to a user process. For example, if the network becomes too congested to accept new messages (or if messages are being sent by a process faster than they are transmitted), STREAM driver queues fill up first. If messages continue flowing from upstream modules, the first module above the driver that has a `service` subroutine will fill up next. This back-pressure potentially propagates all the way up to the STREAM head, which then blocks the user process.

In BSD UNIX, flow control occurs at several locations in the protocol family architecture. The socket level flow control mechanism uses the high- and low-watermarks stored in the socket data structure. If a process tries to `send` more data than is allowed by a socket's highwater mark, the BSD kernel puts the process to sleep. Unlike System V, however, BSD UNIX has no standard mechanism for applying back-pressure between protocol components such as TCP and IP. At the network interface layer, queues are used to buffer messages between the network adapters and the lowest-level protocol (*e.g.,* IP, IDP, or CLNP). The queues have a maximum length that serves as a simple form of flow control. Subsequent incoming messages are dropped if these queues become full.

The *x*-kernel and the Conduit framework provide less systematic flow control support. The *x*-kernel uses a coarse-grained, per-process flow control by discarding incoming messages if there are no light-weight processes available to shepard them up the protocol graph. The Conduit framework does not provide a standard mechanism to manage flow control between modules in a given stack of `Conduits`. Each `Conduit` passes a message up or down to its neighbor. If the neighbor is unable to accept the message, the operation either blocks or returns an error code (in which case the caller may either discard the message or retain it for subsequent retransmission). This approach allows each `Conduit` to determine whether it is a "message-discarding" entity or a "patiently-blocking" entity.

2.5 Summary

This chapter examines the major levels of software mechanisms in the communication subsystem architecture. A taxonomy of six communication subsystem mechanisms is presented and used to compare different design alternatives found in four existing commercial and experimental operating systems. Based upon our experience with communication subsystems, combined with our survey of research literature and existing systems, we view the following as important open research issues pertinent to the development of communication subsystem architectures:

- Which communication subsystem levels (*e.g.,* application interface, session architecture, protocol family architecture, kernel architecture) incur the most communication performance overhead?
- Which choices from among the taxonomy dimensions and alternatives improve the overall communication performance? For example, which process architectures result in the highest performance? Likewise, what combinations of application requirements and network characteristics are most suitable for different communication subsystem profiles?
- How will the performance bottlenecks shift as the boundary between hardware and software changes? For instance, the high cost of message management operations such as fragmentation and reassembly may be greatly reduced if they are performed in hardware, as proposed for ATM.
- Which communication subsystem profiles are best suited for multimedia applications running in high-performance network environments? Moreover, what are the appropriate design strategies and implementation techniques required to provide *integrated* support for multimedia applications that run on general-purpose workstation operating systems?

This chapter attempts to clarify the essential issues and relationships that arise when developing high-performance communication subsystem architectures. Subsequent chapters discuss performance experiments conducted using an object-oriented framework called the ADAPTIVE Service eXecutive (`ASX`). The `ASX` framework supports experimentation with different process architectures for parallelizing high-performance communication subsystem.

Chapter 3

The ADAPTIVE Service eXecutive Framework

3.1 Introduction

Developing extensible, robust, and efficient distributed communication systems is a complex task. To help alleviate this complexity, we have developed the ADAPTIVE Service eXecutive (ASX) framework. ASX is an object-oriented framework composed of automated tools and reusable components that simplify the development, configuration, and reconfiguration, and testing protocol stacks within communication subsystems. The ASX framework has also been used to develop distributed applications in a heterogeneous environment [SS94b, SS94a]. These applications may be configured to contain multiple network services that execute concurrently in one or more processes or threads. Furthermore, the services in these applications may be updated and extended without modifying, recompiling, relinking, or restarting the applications at run-time. This chapter describes the object-oriented architecture of the ASX framework.

3.2 Object-Oriented Frameworks

Object-oriented frameworks help to alleviate the complexity associated with developing, configuring, and reconfiguring distributed application services. A framework is an integrated collection of software components that collaborate to produce a reusable architecture for a family of related applications [JF88]. Object-oriented frameworks are becoming increasingly popular as a means to simplify and automate the development and configuration of complex applications in domains such as graphical user interfaces [LC87, WGM88], databases [BO92], operating system kernels [CRJ87], and communication subsystems [Zwe90, SS94c].

The components in a framework typically include *classes* (such as message managers and timer-based event managers, and connection maps [HP91]), *class hierarchies* (such as an inheritance lattice of mechanisms for local and remote interprocess communication [Sch92]), *class categories* (such as a family of concurrency mechanisms [SS94c]), and *objects* (such as an event demultiplexer [Sch95]).

As shown in Figure 3.1, frameworks are distinguished from conventional class libraries in the following ways:

- The components constituting a framework are integrated together to address a particular problem domain [Sch94a]. In contrast, class library components are often developed to be domain independent. Domain-independent components include class libraries containing mathematical functions and abstract data type components (such as classes for complex numbers, arrays, bitsets, etc.).
- Complete communication systems may be formed by inheriting from and/or customizing existing framework components, rather than invoking methods on objects provided in a class library. Inheritance enables the features of a framework

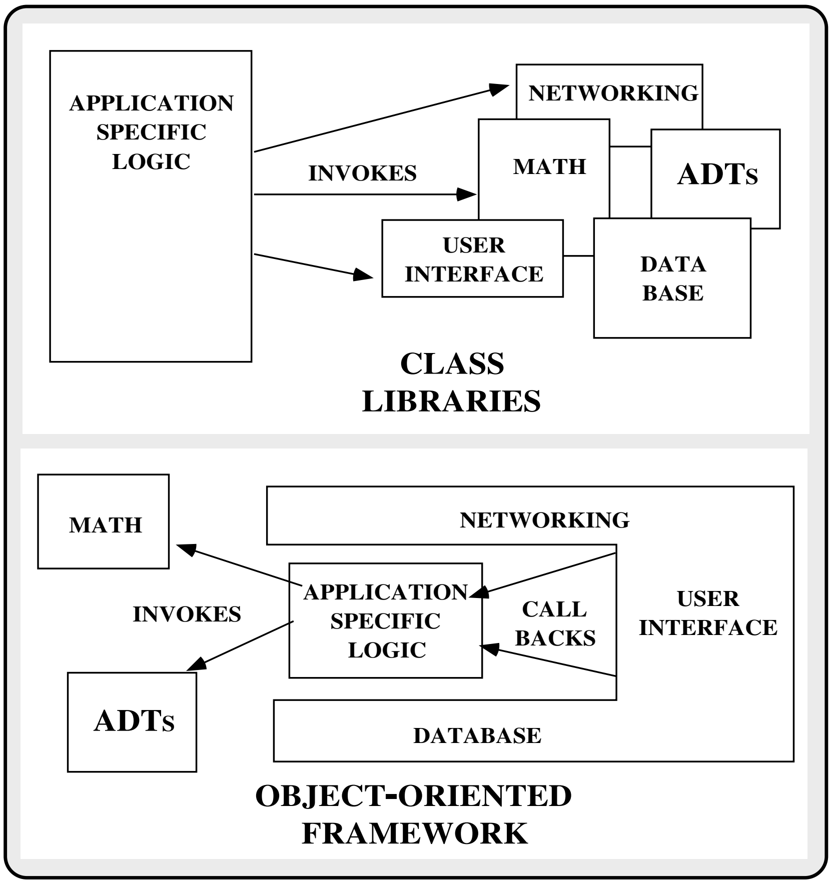

Figure 3.1: Class Libraries vs. Frameworks

class to be shared automatically by its descendant classes. It also allows the framework to be extended transparently without affecting the original code. Developers often interact with an application framework by inheriting basic functionality from its existing scaffolding and overriding certain virtual methods to perform application-specific processing.

- At run-time, the framework is usually responsible for managing the event-loop(s) that provide the default flow of control within an application. The framework determines which set of framework-specific and application-specific methods to invoke in response to external events (such as messages arriving on communication ports).

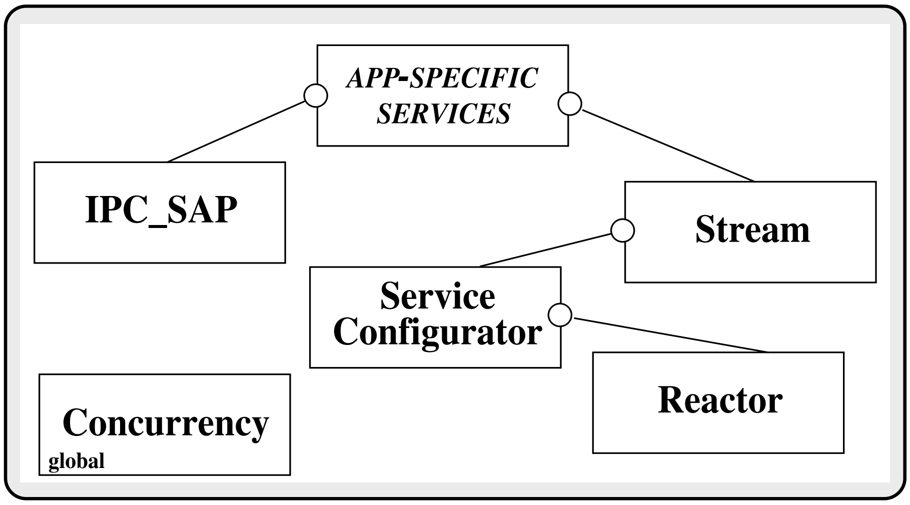

Figure 3.2: Class Categories in the ASX Framework

- By integrating application-specific and application-independent components, frameworks enable larger-scale reuse of software, compared to reusing individual classes and stand-alone functions.

3.3 The Object-Oriented Architectures of the ASX Framework

After prototyping a number of alternative designs, we identified and implemented the class categories illustrated in Figure 3.2. A class category is a collection of components that collaborate to provide a set of related services. A distributed communication system (such as a protocol stack) may be configured by combining components in each of the following class categories via C++ language features such as inheritance, object composition, and template instantiation:

- `Stream` class category – Components in this category are responsible for coordinating the installation-time and run-time configuration of *Stream*s, which are collections of hierarchically-related protocol tasks that are composed to form communication protocol stacks [SS95a].

- The Reactor class category – These components are responsible for *demultiplexing* temporal events generated by a timer-driven callout queue, I/O events received on communication ports, and signal-based events [Sch95, Sch93b, Sch93a]. The Reactor also *dispatches* the appropriate pre-registered handler(s) to process these events.

- The Service Configurator class category – These components are responsible for *dynamically linking* or *dynamically unlinking* services into or out of the address space of an application at run-time [SS94d, SS94a].

- The Concurrency class category – These components are responsible for *spawning, executing, synchronizing*, and *gracefully terminating* services at run-time via one or more threads of control within one or more processes [Sch94b].

- The IPC_SAP class category – These components encapsulate standard OS local and remote IPC mechanisms (such as sockets and TLI) within a type-safe and portable object-oriented interface [Sch92].

Lines connecting the class categories in Figure 3.2 indicate dependency relationships. For example, components that implement the application-specific services in a particular distributed application depend on the Stream components, which in turn depend on the Service Configurator components. Since components in the Concurrency class category are used throughout the application-specific and application-independent portions of the ASX framework they are marked with the **global** adornment.

The ASX framework helps control for several confounding factors such as protocol functionality, concurrency control strategies, application traffic characteristics, and network interfaces. This enables precise measurement of the performance impact from using different process architectures to parallelize protocol stacks in a multi-processor platform. In the experiments described in Chapter 4, the `ASX` framework is used to hold protocol functionality constant, while allowing the process architecture to be systematically altered in a controlled manner.

The `ASX` framework incorporates concepts from several existing communication frameworks such as System V STREAMS [Rit84], the *x*-kernel [HP91], and the Conduit [Zwe90]. These frameworks contain tools that support the flexible configuration of communication subsystems. These tools support the interconnection of building-block protocol components (such as message managers, timer-based event dispatchers, and connection demultiplexers [HP91] and other reusable protocol mechanisms [SSS+93]) to form protocol stacks. In addition to supplying building-block protocol components, the `ASX` framework also extends the features provided by existing communication frameworks. In particular, `ASX` provides components that decouple protocol-specific functionality from the following structural and behavioral characteristics of a communication subsystem:

- The type of locking mechanisms used to synchronize access to shared objects
- The use of message-based and task-based process architectures
- The use of kernel-level vs. user-level execution agents

These `ASX` framework components simplify development of and experimentation with protocol stacks that are functionally equivalent, but possess significantly different process architectures.

The remainder of this section describes the main components in each of the `ASX` framework's class categories. Throughout the dissertation, components in the `ASX` framework are illustrated with Booch notation [Boo93]. Solid clouds indicate objects;

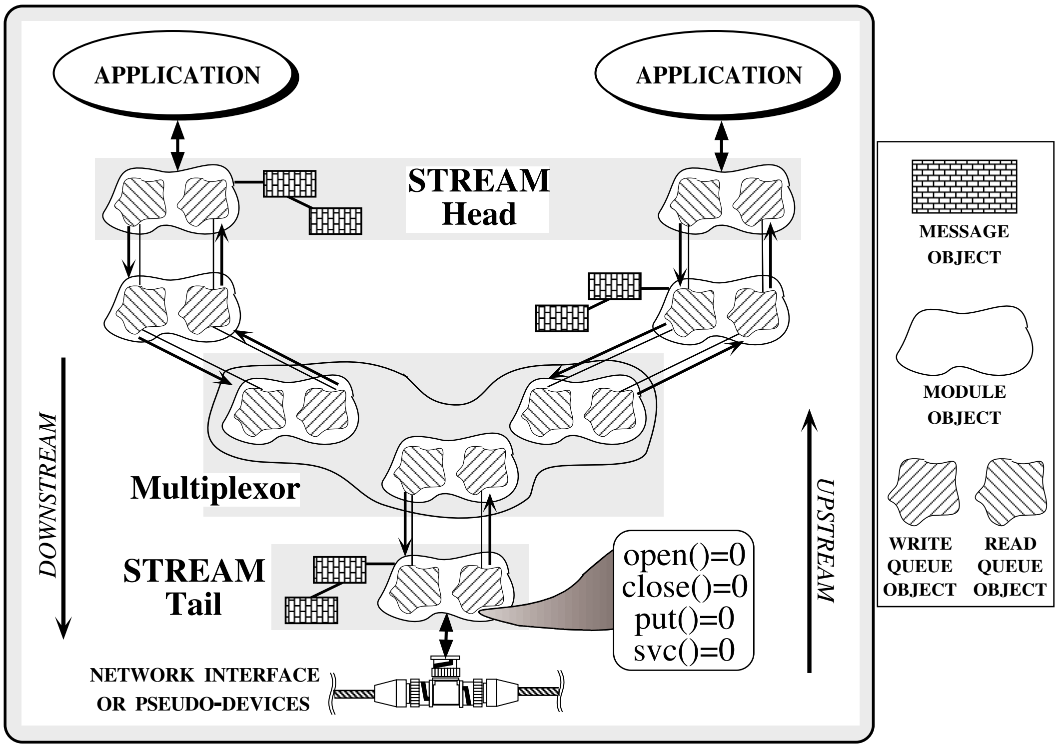

Figure 3.3: Components in the `Stream` Class Category

nesting indicates composition relationships between objects; and undirected edges indicate a link exists between two objects. Dashed clouds indicate classes; directed edges indicate inheritance relationships between classes; and an undirected edge with a small circle at one end indicates either a composition or a uses relation between two classes. Solid rectangles indicate class categories, which combine a number of related classes into a common name space.

3.3.1 The `Stream` Class Category

Components in the `Stream` class category are responsible for coordinating the installation-time and/or run-time configuration of one or more Streams. A Stream is an object that applications communicate with at run-time to configure and execute protocol stacks in the ASX framework. As illustrated in Figure 3.3, a Stream contains a series of interconnected Module objects. Module objects are used to decompose a protocol stack into a functionally distinct levels. Each level implements a cluster of related protocol tasks (such as an end-to-end transport service or a presentation layer formatting service).

Any level that requires multiplexing and/or demultiplexing of messages between one or more related Streams may be developed using a Multiplexor object. A Multiplexor is a container object that provides mechanisms that route messages between Modules in a collection of related Streams. Both Module and Multiplexor objects may be flexibly configured into a Stream by developers at installation-time, as well as by applications at run-time.

Every Module contains a pair of Queue objects that partition a level into the read-side and write-side functionality required to implement a particular protocol task. A Queue provides an abstract domain class that may be specialized to target a specific domain (such as the domain of communication protocol stacks or the domain of network management applications [SS94b]). Each Queue contains a Message_List, which is a reusable ASX framework component that queues a sequence of data messages and control messages for subsequent processing. Protocol tasks in adjacent Modules communicate by exchanging typed messages via a uniform message passing interface defined by the Queue class.

The ASX framework employs a variety of design techniques (such as object-oriented design patterns [Sch95, GHJV94] and hierarchical software decomposition [BO92]) and C++ language features (such as inheritance, dynamic binding, and parameterized types [Bja91]). These design techniques and language features enable developers to flexibly configure protocol-specific functionality into a Stream without modifying the reusable protocol-independent framework components. For example, incorporating a new level of protocol functionality into a Stream at installation-time or at run-time involves the following steps:

1. Inheriting from the `Queue` class interface and selectively overriding several methods (described below) in the new `Queue` subclass to implement protocol-specific functionality
2. Allocating a new `Module` that contains two instances (one for the read-side and one for the write-side) of the protocol-specific `Queue` subclass
3. Inserting the `Module` into a Stream object at the appropriate level (*e.g.,* the transport layer, network layer, data-link layer, etc.)

To avoid reinventing terminology, many component names in the `Stream` class category correspond to similar componentry available in the System V STREAMS framework [Rit84]. However, the techniques used to support extensibility and concurrency in the two frameworks differ significantly. As describe above, incorporating new protocol-specific functionality to an `ASX` Stream is performed by inheriting interfaces and implementations from existing `ASX` framework components. Using inheritance to add protocol-specific functionality provides greater type-safety compared with the pointer-to-function techniques used in System V STREAMS. In addition, the `ASX Stream` classes also redesign and reimplement the co-routine-based, "weightless"[1] service processing mechanisms used in System V STREAMS. These `ASX` changes enable more effective use of multiple PEs on a shared memory multi-processing platform by reducing the opportunities for deadlock and simplifying flow control between `Queues` in a Stream. The remainder of this section discusses the primary components of the `ASX`

[1] A weightless process executes on a run-time stack that is also used by other processes. This complicates programming and increases the potential for deadlock since a weightless process may not suspend execution to wait for resources to become available or for events to occur [SPY+93].

`Stream` class category: the `Stream` class, the `Module` class, the `Queue` class, and the `Multiplexor` class.

3.3.1.1 The STREAM Class

The `STREAM` class defines the application interface to a Stream. A `STREAM` object provides a bi-directional `get`/`put`-style interface that allows applications to access a protocol stack containing a series of interconnected `Modules`. Applications send and receive data and control messages through the stack of `Modules` that comprise a `STREAM` object. In addition, the `STREAM` class implements a `push`/`pop`-style interface that enables applications to configure a Stream at run-time by inserting and removing protocol-specific `Module` class objects.

3.3.1.2 The Module Class

The `Module` class defines a distinct level of protocol functionality in a protocol stack. A Stream is formed incrementally by connecting each `Module` object in the Stream with two adjacent `Module` objects (one "upstream" and one "downstream"). Each `Module` contains a pair of objects that inherit from the `Queue` class described in Section 3.3.1.3 below. A `Module` uses its two `Queue` subclass objects to implement its bi-directional, protocol-specific functionality. A `Module` communicates with its neighboring `Module` objects by passing typed messages. Message passing overhead is minimized by passing a pointer to a message between two `Modules`, rather than by copying the data.

The `Stream_Head` and `Stream_Tail Module` objects shown in Figure 3.3 are installed automatically when a Stream is first opened. These two `Modules` interpret pre-defined `ASX` framework control messages and data messages that pass through a Stream at run-time. In addition, the `Stream_Head Module` provides a synchronous message queueing interface between an application and a Stream. The read-side of a `Stream_Tail Module` transforms incoming messages from a network interface (or from a pseudo-interface such as a loop-back device) into a canonical internal Stream message format. These messages are subsequently processed by higher-level components in a Stream and subsequently delivered to an application. The write-side of a `Stream_Tail Module` transforms outgoing messages from their internal Stream format into the appropriate network message format and passes the message to a network interface.

3.3.1.3 The Queue Abstract Class

The `Queue` abstract class[2] defines an interface that subclasses inherit and selectively override to provide the read-side and write-side protocol functionality in a `Module`. The `Queue` class is an abstract class since its interface defines the four pure virtual methods (`open`, `close`, `put`, and `svc`) described below. By defining `Queue` as an abstract class, the protocol-independent components (such as message objects, message lists, and message demultiplexing mechanisms) provided by the `Stream` class category are decoupled from the protocol-specific subclasses (such as those implementing the data-link, IP, TCP, UDP, and XDR protocols) that inherit and use these components.

[2]An abstract class in C++ provides an interface that contains at least one *pure virtual method* [Bja90]. A pure virtual method provides only an interface declaration, without supplying any accompanying definition for the method. Subclasses of an abstract class must provide definitions for all its pure virtual methods before any objects of the subclass may be instantiated.

This decoupling enhances component reuse and simplifies protocol stack development and configuration.

The `open` and `close` methods in the `Queue` class may be specialized via inheritance to perform activities that are necessary to initialize and terminate a protocol-specific object, respectively. These activities allocate and deallocate resources such as connection control blocks, I/O descriptors, and synchronization locks. When a `Module` is inserted or removed from a Stream, the `ASX` framework automatically invokes the `open` or `close` method of the `Module`'s write-side and read-side `Queue` subclass objects.

The `put` method in a `Queue` is invoked by a `Queue` in an adjacent `Module` passing it a message. The `put` method runs *synchronously* with respect to its caller by borrowing the thread of control from the `Queue` that invoked its `put` method. This thread of control typically originates either upstream from an application, downstream from process(es) that handle network interface device interrupts, or internal to a Stream from an event dispatching mechanism (such as a timer-driven callout queue [BL88] used to trigger retransmissions in a connection-oriented transport protocol).

The `svc` method in a `Queue` may be specialized to perform protocol-specific processing *asynchronously* with respect to other `Queues` in its Stream. A `svc` method is not directly invoked from an adjacent `Queue`. Instead, it is invoked by a separate process associated with the `Queue`. This process provides a separate thread of control that executes the `Queue`'s `svc` method. This method runs an event loop that waits continuously for messages to be inserted into the `Queue`'s `Message_List`. A `Message_List` is a standard component in a `Queue`. Protocol-specific code may reuse the `Message_List` to queue data messages and control messages for subsequent protocol processing. When messages are inserted into a `Message_List`, the

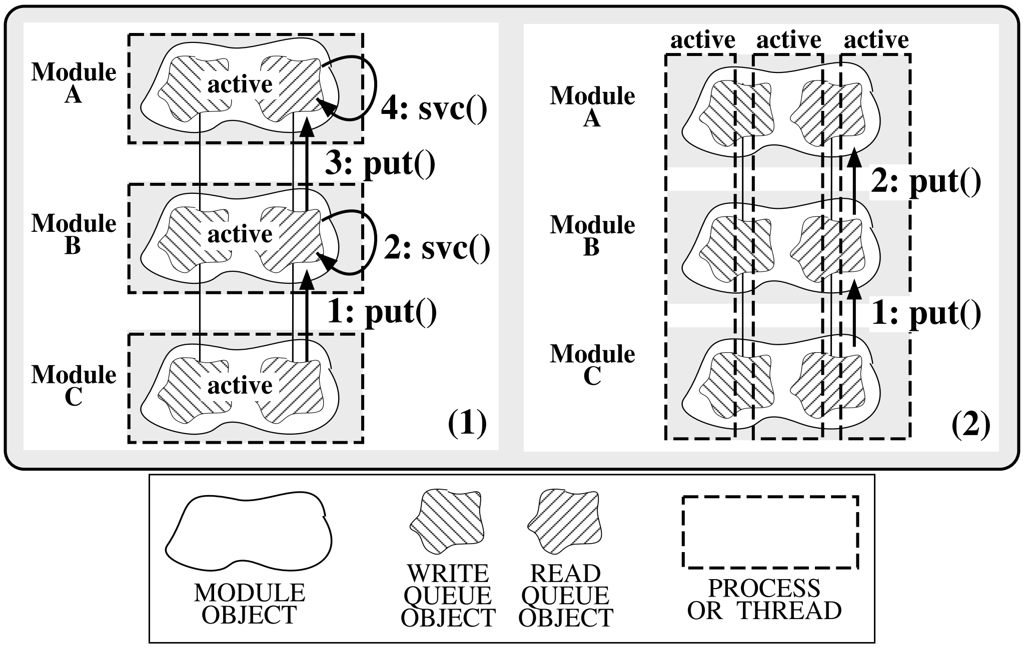

Figure 3.4: Alternative Methods for Invoking `put` and `svc` Methods

`svc` method dequeues the messages and performs the protocol-specific processing tasks defined by the `Queue` subclass.

Within the implementation of a `put` or `svc` method, a message may be forwarded to an adjacent `Queue` in a Stream via the `Queue::put_next` method. `Put_next` calls the `put` method of the next `Queue` residing in an adjacent `Module`. This invocation of `put` may borrow the thread of control from its caller and process the message immediately (*i.e.,* the synchronous processing approach illustrated in Figure 3.4 (1)). Conversely, the `put` method may enqueue the message and defer processing to its `svc` method, which executes in a separate thread of control (*i.e.,* the asynchronous processing approach illustrated in Figure 3.4 (2)).

In general, message-based process architectures perform their protocol-specific processing *synchronously* within the `put` methods of their `Queues`. For instance, in the `ASX`-based implementation of the Message Parallelism process architecture, a message arriving at a network interface is associated with a separate thread of control. This thread of control is obtained from a pool of pre-initialized processes (which are labeled as the active objects in Figure 3.4 (1)). The incoming message is escorted through a series of interconnected `Queues` in a Stream. A synchronous upcall [Cla85] to the `put` method in an adjacent `Queue` is performed to pass a message to each higher level in a protocol stack.

Task-based process architectures perform their processing *asynchronously* in the `svc` methods of their `Queues`, which execute in separate threads of control. For instance, each protocol layer in the `ASX`-based Layer Parallelism process architecture is implemented in a separate `Module` object, which is associated with a separate process. Messages arriving from a network interface are passed between `Queues` running in the separate processes (which are labeled as the active objects in Figure 3.4 (2)). Each `Queue` in a Stream asynchronously executes its protocol-specific tasks within its `svc` method. Chapter 4 illustrates the performance impact of using synchronous vs. asynchronous processing to parallelize communication protocol stacks on a shared memory multi-processor platform.

3.3.1.4 The Multiplexor Class

A `Multiplexor` routes messages between different Streams that implement layered protocol stacks. Although layered multiplexing and demultiplexing is generally disparaged for high-performance communication subsystems [Fel90], most conventional communication models (such as the Internet model or the ISO/OSI reference model) require some form of multiplexing. Thus, the `ASX` framework provides mechanisms that support it.

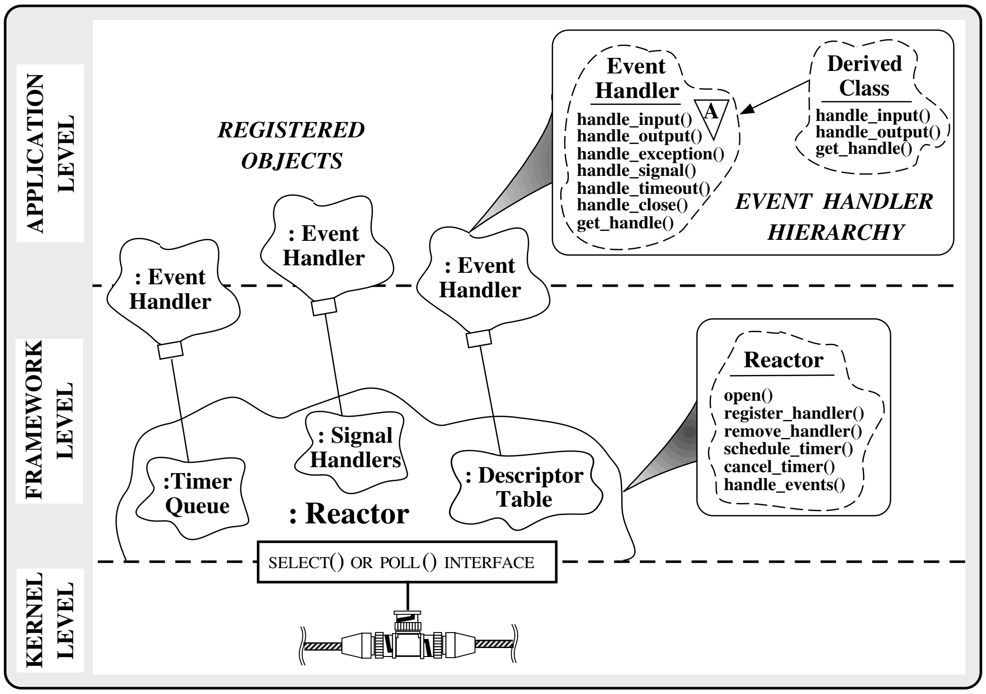

Figure 3.5: Components in the `Reactor` Class Category

A `Multiplexor` is implemented via a C++ template class called `Map_Manager`. The `Map_Manager` class is parameterized by an external identifier type (which serves as the search key into a map) and an internal identifier type (which contains the information associated with each search key). Protocol-specific `Multiplexors` may be formed by instantiating particular external and internal identifier type parameters into the `Map_Manager` template class. This instantiation produces specialized objects that perform efficient intra-Stream message routing. Common external identifiers used in protocol stacks include network addresses, port numbers, or type-of-service fields. Likewise, common internal identifiers include pointers to `Modules` or pointers to protocol connection records.

3.3.2 The `Reactor` Class Category

Components in the `Reactor` class category [Sch95, Sch93b, Sch93a] are responsible for demultiplexing I/O-based events received on communication ports, time-based events generated by a timer-driven callout queue, or signal-based events. When these events occur at run-time, the `Reactor` dispatches the appropriate pre-registered handler(s) to process the events. The `Reactor` encapsulates and enhances the functionality of OS event demultiplexing mechanisms (such as the UNIX `select` and `poll` system calls or the Windows NT `WaitForMultipleObjects` system call). These OS mechanisms detect the occurrence of different types of input and output events on one or more I/O descriptors simultaneously.

The `Reactor` contains the methods illustrated in Figure 3.5. These methods provide a uniform interface to manage objects that implement various types of application-specific handlers. Certain methods register, dispatch, and remove I/O descriptor-based and signal-based handler objects from the `Reactor`. Other methods schedule, cancel, and dispatch timer-based handler objects. As shown in Figure 3.5, these handler objects all derive from the `Event_Handler` abstract base class. This class specifies an interface for event registration and service handler dispatching.

The `Reactor` uses the virtual methods defined in the `Event_Handler` interface to demultiplex I/O descriptor-based, timer-based, and signal-based events. When these events occur at run-time, the `Reactor` dispatches the appropriate pre-registered handler(s) to process the events. I/O descriptor-based events are dispatched via the `handle_input`, `handle_output`, and `handle_exception` methods; timer-based events are dispatched via the `handle_timeout` method; and Signal-based events are dispatched via the `handle_signal` method. Subclasses of `Event_Handler` may augment the base class interface by defining additional methods and data members. In addition, virtual methods in the `Event_Handler` interface may be selectively overridden to implement application-specific functionality. Once the pure virtual methods in

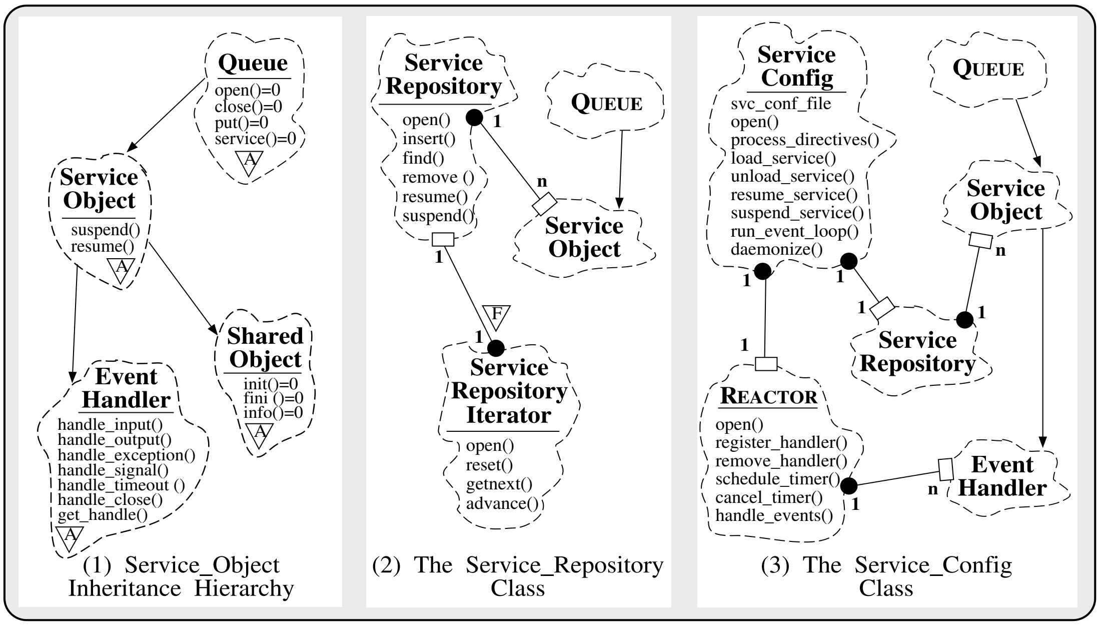

Figure 3.6: Components in the `Service Configurator` Class Category

the `Event_Handler` base class have been supplied by a subclass, an application may define an instance of the resulting composite service handler object.

When an application instantiates and registers a composite I/O descriptor-based service handler object, the `Reactor` extracts the underlying I/O descriptor from the object. This descriptor is stored in a table along with I/O descriptors from other registered objects. Subsequently, when the application invokes its main event loop, these descriptors are passed as arguments to the underlying OS event demultiplexing system call (*e.g.,* `select`, `poll`, or `WaitForMultipleObjects`). As events associated with a registered handler object occur at run-time, the `Reactor` automatically detects these events and dispatches the appropriate method(s) of the service handler object associated with the event. This handler object then becomes responsible for performing its application-specific functionality before returning control to the main `Reactor` event-loop.

3.3.3 The `Service Configurator` Class Category

Components in the `Service Configurator` class category are responsible for dynamically linking or unlinking protocol tasks into or out of the address space of a communication subsystem at run-time. Dynamic linking enables the configuration and reconfiguration of protocol-specific services *without* requiring the modification, recompilation, relinking, or restarting of an executing system. The `Service Configurator` components discussed below include the the `Service_Object` inheritance hierarchy (Figure 3.6 (1)), the `Service_Repository` class (Figure 3.6 (2)), and the `Service_Config` class (Figure 3.6 (3)). This discussion focuses on using the `Service Configurator` class category in the context of configuring application-tailored protocol stacks [SSS+93]. However, these components are also useful in other domains such as network management [SS94b] and configurable distributed systems [SS94a].

3.3.3.1 The Service_Object Inheritance Hierarchy

The `Service_Object` class is the focal point of a multi-level hierarchy of types related by inheritance. The interfaces provided by the abstract classes in this type hierarchy may be selectively implemented by application-specific subclasses to access `Service Configurator` features. These features provide transparent dynamic linking, service handler registration, event demultiplexing, service dispatching, and service run-time control (such as suspending and resuming a service temporarily). By decoupling the application-specific portions of a handler from the underlying `Service Configurator` mechanisms, the effort necessary to insert and remove services from an application at run-time is reduced significantly.

The Service_Object inheritance hierarchy consists of the Event_Handler and Shared_Object abstract base classes, as well as the Service_Object abstract derived class. The Event_Handler class was described above in the Reactor Section 3.3.2. The behavior of the other classes in the Service Configurator class category is outlined below:

• **The Shared_Object Abstract Base Class:** This abstract base class specifies an interface for dynamically linking and unlinking objects into and out of the address space of an application. This abstract base class exports three pure virtual methods: init, fini, and info. These methods impose a compiler-enforced contract between the reusable, service-independent components provided by the Service Configurator and application-specific objects that utilize these components. The use of pure virtual methods enables the Service Configurator to ensure that a service handler implementation honors its obligation to provide certain configuration-related information. This information is subsequently used by the Service Configurator to automatically link, initialize, identify, and unlink a service at run-time.

The init method serves as the entry-point to an object during run-time initialization. This method performs application-specific initialization when an object derived from Shared_Object is dynamically linked. The info method returns a humanly-readable string that concisely reports service addressing information and documents service functionality. Clients may query an application to retrieve this information and use it to contact a particular service running in the application. The fini method is called automatically by the Service Configurator class category when an object is unlinked and removed from an application at run-time. This method typically performs termination operations that release dynamically allocated resources (such as memory or synchronization locks).

The `Shared_Object` class is defined independently from the `Event_Handler` class to clearly separate their two orthogonal sets of concerns. For example, certain applications (such as a compiler or text editor) might benefit from dynamic linking, though they might not require I/O descriptor-based, timer-based, signal-based event demultiplexing. Conversely, other applications (such as an `ftp` server) require event demultiplexing, but might not require dynamic linking.

• **The Service_Object Abstract Derived Class:** Support for dynamic linking, event demultiplexing, and service dispatching is necessary to automate the dynamic configuration and reconfiguration of application-specific services in a distributed system. Therefore, the `Service Configurator` class category provides the `Service_Object` class, which combines the interfaces inherited from both the `Event_Handler` and the `Shared_Object` abstract base classes. During development, application-specific subclasses of `Service_Object` may implement the `suspend` and `resume` virtual methods in this class. The `suspend` and `resume` methods are invoked automatically by the `Service Configurator` class category in response to certain external events (such as those triggered by receipt of the UNIX SIGHUP signal). An application developer may define these methods to perform actions necessary to suspend a service object without unlinking it completely, as well as to resume a previously suspended service object. In addition, application-specific subclasses must implement the four pure virtual methods (`init`, `fini`, `info`, and `get_handle`) that are inherited (but not defined) by the `Service_Object` subclass.

Note that the `Queue` class in the `Stream` class category (described in Section 3.3.1) is derived from the `Service_Object` inheritance hierarchy (illustrated in Figure 3.6 (1)). This enables hierarchically-related, application-specific `Queues` (which are grouped together to form the Modules comprising an application Stream) to be linked and unlinked into and out of an application at run-time.

3.3.3.2 The Service_Repository Class

The ASX framework supports the configuration of applications that contain one or more Streams, each of which may have one or more interconnected application-specific Modules. Therefore, to simplify run-time administration, it may be necessary to individually and/or collectively control and coordinate the Service_Objects that comprise an application's currently active Streams. The Service_Repository is an object manager that the ASX framework uses to coordinate local/remote queries and updates regarding the services offered by Streams in an application. A search structure within the object manager binds service names (represented as a string) with instances of composite Service_Objects (represented as C++ object code). A service name uniquely identifies an instance of a Service_Object stored in the repository.

Each Service_Repository entry contains a pointer to the Service_Object portion of an application-specific subclass (shown in Figure 3.6 (2)). This enables the Service Configurator classes to automatically load, enable, suspend, resume, or unload Service_Objects from a Stream dynamically. The repository also maintains a handle to the shared object file for each dynamically linked Service_Object. This handle is used to unlink and unload a Service_Object from a running application when its service is no longer required. An iterator class is also supplied along with the Service_Repository. This class may be used to visit every Service_Object in the repository without compromising data encapsulation.

```
<svc-config-entries> ::= svc-config-entries
                         svc-config-entry | NULL
<svc-config-entry> ::= <dynamic> | <static> | <suspend>
                   | <resume> | <remove> | <stream> | <remote>
<dynamic> ::= DYNAMIC <svc-location> [ <parameters-opt> ]
<static> ::= STATIC <svc-name> [ <parameters-opt> ]
<suspend> ::= SUSPEND <svc-name>
<resume> ::= RESUME <svc-name>
<remove> ::= REMOVE <svc-name>
<stream> ::= STREAM <stream_ops> '{' <module-list> '}'
<stream_ops> ::= <dynamic> | <static>
<remote> ::= STRING '{' <svc-config-entry> '}'
<module-list> ::= <module-list> <module> | NULL
<module> ::= <dynamic> | <static> | <suspend>
           | <resume> | <remove>
<svc-location> ::= <svc-name> <type> <svc-initializer>
                   <status>
<type> ::= SVC_OBJECT '*' | MODULE '*' STREAM '*' | NULL
<svc-initializer> ::= <object-name> | <function-name>
<object-name> ::= PATHNAME ':' IDENT
<function-name> ::= PATHNAME ':' IDENT '(' ')'
<status> ::= ACTIVE | INACTIVE | NULL
<parameters-opt> ::= STRING | NULL
```

Figure 3.7: EBNF Format for a Service Config Entry

3.3.3.3 The Service_Config Class

As illustrated in Figure 3.6 (3), the `Service_Config` class integrates several other `ASX` framework components (such as the `Reactor`, the `Service_Repository`, the `Service_Object` inheritance hierarchy). Applications use the `Service_Config` class to automate the static and/or dynamic configuration of hierarchically-related network services to form one or more Streams. A configuration file (known as `svc.conf`) is used to guide the configuration and reconfiguration activities of a `Service_Config` object. This configuration file is specified using a scripting language, whose syntactical structure is shown using extended-Backus/Naur Format (EBNF) in Figure 3.7.

The `svc.conf` file may be used to configure one or more Streams into an application. Each entry in the file begins with a directive (such as `dynamic`, `remove`, suspend, or resume) that specifies which configuration activity to perform. Each entry also contains attributes that indicate the location of the shared object file for each dynamically linked service, as well as any parameters required to initialize a service when it is linked at run-time. By consolidating service attributes and initialization parameters into a single configuration file, the installation and administration of the services in an application is simplified. In addition, the svc.conf file helps to decouple the structure of an application from the behavior of its services. This decoupling separates the service-independent configuration and reconfiguration of mechanisms provided by the framework from the application-specific attributes and parameters specified in the svc.conf file.

Figure 3.8 illustrates a state transition diagram depicting the Service_Config and Reactor methods that are invoked in response to events occurring during service configuration, execution, and reconfiguration. For example, when the CONFIGURE and RECONFIGURE events occur, the process_directives method of the Service_Config class is called to parse the svc.conf file associated with the application. The contents of a svc.conf file dictate the content of Streams that are configured into an application. This file is first parsed when a new instance of a daemon is initially configured. The file is parsed again whenever a daemon reconfiguration is triggered upon receipt of a pre-designated external event (such as the UNIX SIGHUP signal).

3.3.4 The Concurrency Class Category

Components in the Concurrency class category are responsible for spawning, executing, synchronizing, and gracefully terminating services via one or more threads of control at run-time. These threads of control execute protocol tasks and pass messages

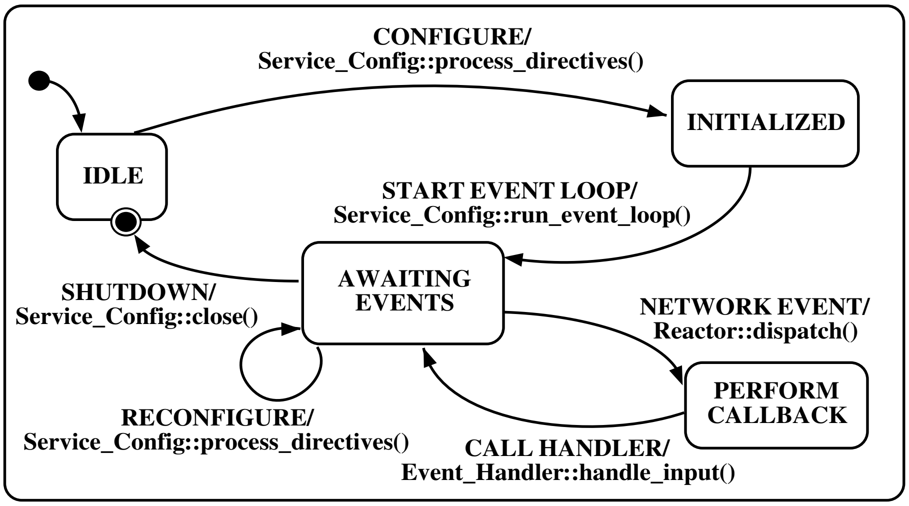

Figure 3.8: State Transition Diagram for Service Configuration, Execution, and Reconfiguration

between `Modules` in a protocol stack. The following section outlines the classes in the `Concurrency` class category.

3.3.4.1 The Synch Classes

Components in the `Stream` class category described in Section 3.3.1 contain minimal internal locking mechanisms. By default, synchronization mechanisms from the `Synch` classes are only used to protect the `ASX` framework components that would not function correctly in a preemptive, multi-threaded parallel processing environment. This design strategy avoids over-constraining the granularity of a process architecture's concurrency control policies.

The `Synch` classes provide type-safe C++ interfaces for two basic types of synchronization mechanisms: `Mutex` and `Condition` objects [Bir89]. A `Mutex` object is used to ensure the integrity of a shared resource that may be accessed concurrently by multiple processes. A `Condition` object allows one or more cooperating processes to suspend their execution until a condition expression involving shared data attains a particular state.

A `Mutex` object serializes the execution of multiple processes by defining a critical section where only one thread of control may execute its code at a time. To enter a critical section, a process invokes the `Mutex::acquire` method. When a process leaves its critical section, it invokes the `Mutex::release` method. These two methods are implemented via adaptive spin-locks that ensure mutual exclusion by using an atomic hardware instruction. An adaptive spin-lock polls a designated memory location using the atomic hardware instruction until one of the following conditions occur:

- The value at this location is changed by the process that currently owns the lock. This signifies that the lock has been released and may now be acquired by the spinning process.
- The process that is holding the lock goes to sleep. At this point, the spinning process also puts itself to sleep to avoid unnecessary polling [EKB+92].

On a multi-processor, the overhead incurred by a spin-lock is relatively minor. Hardware-based polling does not cause contention on the system bus since it only affects the local PE caches of processes that are spinning on a `Mutex` object.

A `Condition` object provides a different type of synchronization mechanism. Unlike the adaptive spin-lock `Mutex` objects, a `Condition` object enables a process to suspend itself indefinitely (via the `Condition::wait` method) until a condition expression involving shared data attains a particular state. When another cooperating process indicates that the state of the shared data has changed (by invoking the `Condition::signal` method), the associated `Condition` object wakes up a process that is suspended on that `Condition` object. The newly awakened process then re-evaluate its condition expression and potentially resumes processing if the shared data has attained an appropriate state.

`Condition` object synchronization is not implemented using spin-locks. Spin-locks consume excessive resources if a process must wait an indefinite amount of time for a particular condition to become signaled. Therefore, `Condition` objects are implemented via sleep-locks that trigger a context switch to allow another process to execute. Chapter 4 illustrates the consequences of context switching and synchronization on process architecture performance.

The `ASX` framework enables the concurrency control strategies used by different process architectures to be selected by instrumenting protocol tasks with various combinations of `Mutex` and `Condition` synchronization objects. When used in conjunction with C++ language features such as parameterized types, these synchronization objects help to decouple protocol processing functionality from the concurrency control strategy used by a particular process architecture [Sch94d]. An illustration of how `ASX` framework synchronization objects are transparently parameterized into communication protocol code is presented in the following example.

In the process architecture experiments described in Chapter 4, the `Map_Manager` class is used to demultiplex incoming network messages to the appropriate `Module`. As discussed in Section 3.3.1.4, `Map_Manager` is a template class that is parameterized by an external identifier (`EXT_ID`), an internal identifier (`INT_ID`), and a mutual exclusion mechanism (`MUTEX`), as follows:

```
template <class EXT_ID, class INT_ID, class MUTEX>
class Map_Manager
{
public:
  // Associate EXT_ID with INT_ID
  bool bind (EXT_ID, INT_ID *);
  // Break an association
  bool unbind (EXT_ID);
```

```
  // Locate INT_ID corresponding to EXT_ID
  bool find (EXT_ID, INT_ID &);
  // ...

private:
  // Parameterized synchronization object
  MUTEX lock;
  // Perform the lookup
  bool locate_entry (EXT_ID, INT_ID &);
};
```

The `find` method of the `Map_Manager` template class is implemented using the technique illustrated in the code below (the `bind` and `unbind` methods are implemented in a similar manner):

```
template <class EXT_ID, class INT_ID, class MUTEX>
bool Map_Manager<EXT_ID, INT_ID, MUTEX>
     ::find (EXT_ID ext_id, INT_ID &int_id)
{
  // Acquire lock in mon constructor
  Mutex_Block<MUTEX> mon (this->lock);

  if (this->locate_entry (ext_id, int_id))
    return true;
  else
    return false;
  // Release lock in mon destructor
}
```

The code shown above uses the constructor of the `Mutex_Block` class to acquire the `Map_Manager` lock when an object of the class is created. Likewise, when the `find` method returns, the destructor for the `Monitor` object releases the `Mutex` lock. Note that the `Mutex` lock is released automatically, regardless of which arm in the `if/else` statement returns from the `find` method. Moreover, the lock is released even if an exception is raised within the body of the `find` method.

The experiments described in Chapter 4 implement connection demultiplexing operations in the connection-oriented protocol stacks via the `Map_Manager` template class. In the experiments, the `Map_Manager` class is instantiated with a `MUTEX` parameter whose type is determined by the process architecture being configured. For instance, the Map_Manager used in the Message Parallelism implementation of the transport layer in the protocol stack described in Section 4.5.1 is instantiated with the following EXT_ID, INT_ID, and MUTEX type parameters:

```
typedef Map_Manager <TCP_Addr, TCB, Mutex> ADDR_MAP;
```

This particular instantiation of Map_Manager locates the transport control block (TCB) internal identifier associated with the address of an incoming TCP message (TCP_Addr) external identifier. Instantiating the Map_Manager class with the Mutex class ensures that its find method executes as a critical section. This prevents race conditions from occurring with other threads of control that are inspecting or inserting entries into the Map_Manager in parallel.

In contrast, the Layer Parallelism implementation of the transport layer in the protocol stack described in Section 4.5.2 uses a different type of concurrency control. In this case, serialization is performed at the transport layer using the synchronization mechanisms provided by the Message_List defined in the Queue class. Therefore, the Map_Manager used for the Layer Parallelism implementation of the protocol stack is instantiated with a different Synch class, as follows:

```
typedef Map_Manager <TCP_Addr, TCB, Null_Mutex> ADDR_MAP;
```

The implementation of the acquire and release methods in the Null_Mutex class are "no-op" inline functions that are removed completely by the compiler optimizer. In general, templates generate efficient object code that exacts no additional run-time overhead for the increased flexibility.

The definition of the Map_Manager address map may be conditionally compiled using template class arguments corresponding to the type of process architecture that is required, *i.e.*:

```
typedef
// Select a message-based process architecture
#if defined (MSG_BASED_PA)
Map_Manager <TCP_Addr, TCB, Mutex>
// Select a task-based process architecture
#elif defined (TASK_BASED_PA)
Map_Manager <TCP_Addr, TCB, Null_Mutex>
#endif
ADDR_MAP;
```

As shown below, this allows the majority of the protocol code to remain unaffected, regardless of the choice of process architecture, as follows:

```
ADDR_MAP  addr_map;
TCP_Addr  tcp_addr;
TCP       tcb;

// ...

if (addr_map.find (tcp_addr, tcb))
     // Perform connection-oriented processing
```

3.3.4.2 The Thread_Manager Class

The `Thread_Manager` class contains a set of mechanisms that manipulate multiple threads of control atomically. Typically, these threads of control collaborate to implement collective actions (such as rendering different portions of a large image in parallel). The `Thread_Manager` class also shields applications from non-portable incompatibilities between different flavors of multi-threading mechanisms (such as POSIX threads, MACH cthreads, Solaris threads, and Windows NT threads).

The `Thread_Manager` class provides methods (such as `suspend_all` and `resume_all`) that suspend and resume a set of collaborating threads of control atomically. This feature is useful for protocol stacks that execute multiple tasks or process multiple messages in parallel. For example, a Stream implemented using the Layer Parallelism process architecture is composed of `Modules` that execute in separate threads

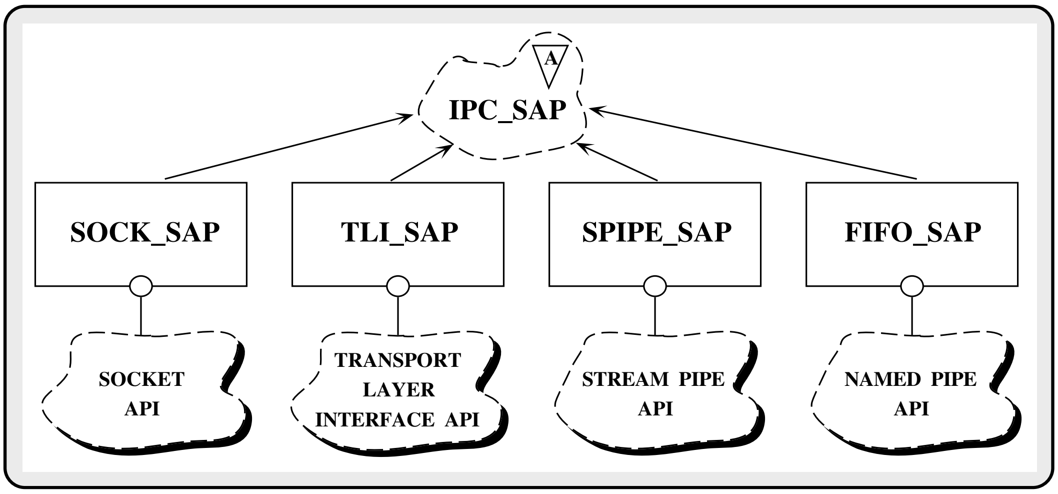

Figure 3.9: IPC_SAP Class Category Relationships

of control. It is crucial that all Modules in the Stream are completely interconnected before allowing messages to be passed between Queues in the Stream. The mechanisms in the Thread_Manager class allow these initialization activities to execute atomically.

3.3.5 The IPC_SAP Class Category

ACE provides a forest of class categories rooted at the IPC_SAP ("InterProcess Communication Service Access Point") base class. IPC_SAP encapsulates the standard I/O descriptor-based operating system local and remote IPC mechanisms that offer connection-oriented and connectionless protocols. As shown in Figure 3.9, this forest of class categories includes SOCK_SAP (which encapsulates the socket API), TLI_SAP (which encapsulates the TLI API), SPIPE_SAP (which encapsulates the UNIX System V release 4 STREAM pipe API), and FIFO_SAP (which encapsulates the UNIX named pipe API).

Each class category is organized as an inheritance hierarchy. Every subclass provides a well-defined interface to a subset of local or remote communication 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). The use of classes (as opposed to stand-alone functions) helps to simplify network programming in the following manner:

- *Reduce potential for programmer error* – For example, the `Addr` class hierarchy shown in Figure 3.9 supports several diverse network addressing formats via a type-secure C++ interface, rather than using the awkward and error-prone C-based `struct sockaddr` data structures directly.
- *Combining several operations to form a single operation* – For example, the constructor in the `SOCK_Listener` class performs the various socket system calls (such as `socket`, `bind`, and `listen`) required to create a passive-mode server endpoint.
- *Parameterizing IPC mechanisms into applications* – Classes form the basis for parameterizing an application by the type of IPC mechanism it requires. This helps to improve portability as discussed in Section 3.3.5.2.
- *Enhance code sharing* – Inheritance-based hierarchical decomposition increases the amount of common code that is shared amongst the various IPC mechanisms (such as the C++ interface to the lower-level UNIX operating system device control system calls like `fcntl` and `ioctl`).

The following sections discuss each of the class categories in `IPC_SAP`.

3.3.5.1 SOCK_SAP

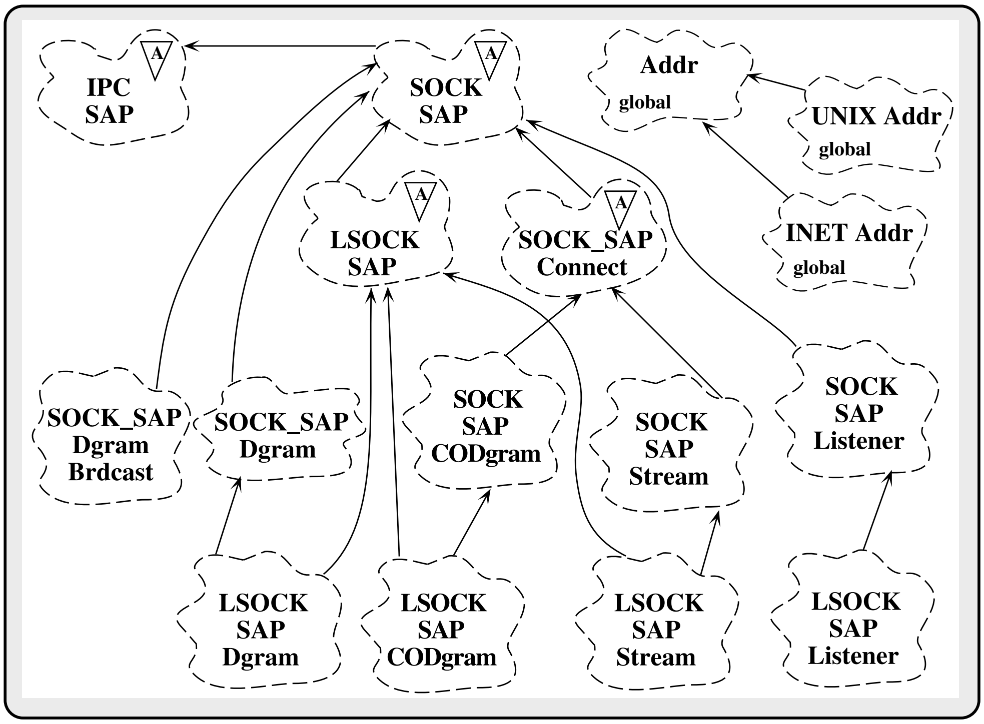

Figure 3.10: The SOCK_SAP Inheritance Hierarchy

The SOCK_SAP [Sch92] C++ class category provides applications with an object-oriented interface to the Internet-domain and UNIX-domain protocol families [Ste90]. 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 3.10. The SOCK* subclasses encapsulate Internet-domain functionality and the LSOCK* subclasses encapsulate UNIX-domain functionality. As shown in Figure 3.10, the subclasses may be further decomposed into (1) the *Dgram components (which provide unreliable, connectionless, message-oriented functionality) vs. the *Stream components (which provide reliable, connection-oriented, bytestream functionality) and (2) the *Listener components (which provide connection establishment functionality typically used by servers) vs. the *Stream components (which provide bi-directional bytestream data transfer functionality used by both clients and servers).

Using C++ wrappers to encapsulate the socket interface helps to (1) detect many 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 that uses the SOCK_Dgram_Brdcast class to broadcast a message to all servers listening on a designated port number in a LAN subnet::

```
int
main (int argc, char *argv[])
{
  SOCK_Dgram_Brdcast b_sap (sap_any);
  char *msg;
  unsigned 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"), exit (1);
  exit (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.

3.3.5.2 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 straightforward 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. 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_IO header file */
template <class XPORT_SAP, class ADDR>
class Logging_IO : public Event_Handler
{
public:
  Logging_IO (void);
  virtual ~Logging_IO (void);

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

protected:
  XPORT_SAP 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_IO` class to use either `SOCK_SAP` or `TLI_SAP`, as shown below:

```
/* Logging application */
class Logging_IO : public
#if defined (MT_SAFE_SOCKETS)
Logging_IO<SOCK_Stream, INET_Addr>
#else
Logging_IO<TLI_Stream, INET_Addr>
#endif /* MT_SAFE_SOCKETS */
{
  /* ... */
};
```

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 subtle application-level code required to properly handle the non-intuitive, error-prone behavior of `t_listen` and `t_accept` in a concurrent server with a `qlen` > 1 [Rag93] is encapsulated within the `accept` method in the `TLI_Listener` class. This method accepts incoming connection requests from clients. 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.

3.3.5.3 SPIPE_SAP

The `SPIPE_SAP` class category provides a C++ wrapper interface for mounted STREAM pipes and `connld` [PR90]. SunOS 5.x provides the `fattach` system call that mounts a pipe descriptor at a designated location in the UNIX file system. A server application is created by pushing the `connld` STREAM module onto the mounted end of the pipe. When a client application running on the same host machine as the server subsequently opens the filename associated with the mounted pipe, the client and server each receive an I/O descriptor that identifies a unique, non-multiplexed, bi-directional channel of communication.

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` `LSOCK*` classes (which themselves encapsulate UNIX-domain sockets). However, SPIPE_SAP is more flexible than the 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 byte-stream and prioritized message-oriented data between processes and/or threads executing within the same host machine [Ste92].

3.3.5.4 FIFO_SAP

The FIFO_SAP class category encapsulates the UNIX named pipe mechanism (also called FIFOs). Unlike STREAM pipes, named pipes 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 named pipes in System V release 4 provides both message-oriented and bytestream-oriented data delivery semantics. In contrast, SunOS 4.x only provides bytestream-oriented named pipes. Therefore, unless fixed length messages are always used, each message sent via a named pipe in SunOS 4.x must be distinguished by some form of byte count or special termination symbol that allows a receiver to extract messages from the communication channel bytestream. To alleviate this limitation, the SunOS 4.x FIFO_SAP implementation contains logic that emulates the message-oriented semantics available in System V release 4.

3.4 Summary

Despite an increase in the availability of operating system and hardware platforms that support networking and parallel processing [EKB+92, Gar90, SPY+93, TRG+87, Cus93], developing distributed and parallel communication systems remains a complex and challenging task. The ADAPTIVE Service eXecutive (`ASX`) provides an extensible object-oriented framework that simplifies the development of distributed applications on multi-processor platforms. The `ASX` framework employs a variety of advanced OS mechanisms (such as multi-threading and explicit dynamic linking), object-oriented design techniques (such as encapsulation, hierarchical classification, and deferred composition) and C++ language features (such as parameterized types, inheritance, and dynamic binding) to enhance software quality factors (such as robustness, ease of use, portability, reusability, and extensibility) without degrading application performance. In general, the object-oriented techniques and C++ features enhance the software quality factors, whereas the advanced OS mechanisms improve application functionality and performance.

The `ASX` framework components described in this chapter are freely available via anonymous ftp from `ics.uci.edu` in the file `gnu/C++_wrappers.tar.Z`. This distribution contains complete source code, documentation, and example test drivers for the C++ components developed as part of the ADAPTIVE project [SBS93, BSS93] at the University of California, Irvine. Components in the `ASX` framework have been ported to both UNIX and Windows NT and have been used in a number of commercial products including the AT&T Q.port ATM signaling software product, the Ericsson EOS family of PBX monitoring applications, and the network management portion of the Motorola Iridium mobile communications system.

Chapter 4

Communication Subsystem Performance Experiments

4.1 Introduction

This chapter presents performance results obtained by measuring the data reception portion of protocol stacks implemented using several process architectures developed using the `ASX` framework [SS95b]. Two different types of protocol stacks were implemented: *connection-oriented* and *connectionless*. Three different variants of task-based and message-based process architectures were used to parallelize the protocol stacks: *Layer Parallelism* (which is a task-based process architecture), as well as *Message-Parallelism* and *Connectional Parallelism* (which are message-based process architectures). This chapter outlines related work on the performance of parallel process architectures. It also describes the multi-processor platform and the measurement tools used in the experiments, as well as the communication protocol stacks and process architectures developed using `ASX` framework components. Finally, the performance results from the experiments are presented and analyzed.

4.2 Related Work

A number of studies have investigated the performance characteristics of task-based process architectures that ran on either message passing or shared memory platforms. [WF93] measured the performance of several implementations of the transport and session layers in the ISO OSI reference model using an ADA-like rendezvous-style of Layer Parallelism in a nonuniform access shared memory multi-processor platform. [LKAS93] measured the performance of Functional Parallelism for presentation layer and transport layer functionality on a shared memory multi-processor platform. [BZ93] measured the performance of a de-layered, function-oriented transport system [ZST93] using Functional Parallelism on a message passing transputer multi-processor platform. An earlier study [Zit91] measured the performance of the ISO OSI transport layer and network layer, also on a transputer platform. Likewise, [GKWW89] used a multi-processor transputer platform to parallelize several data-link layer protocols.

Other studies have investigated the performance characteristics of message-based process architectures. All these studies utilized shared memory platforms. [HP91] measured the performance of the TCP, UDP, and IP protocols using Message Parallelism on a uniprocessor platform running the *x*-kernel. [Mat93] measured the impact of synchronization on Message Parallelism implementations of TCP and UDP transport protocols built within a multi-processor version of the *x*-kernel. Likewise, [NYKT94] examined performance issues in parallelizing TCP-based and UDP-based protocol stacks using a different multi-processor version of the *x*-kernel. [Pre93] measured the performance of the Nonet transport protocol on a multi-processor version of Plan 9 STREAMS developed using Message Parallelism. [GNI92] measured the performance of the ISO OSI protocol stack, focusing primarily on the presentation and transport layers using Message Parallelism. [SPY+93] measured the performance of the TCP/IP protocol stack using Connectional Parallelism in a multi-processor version of System V STREAMS.

The research presented in this paper extends existing work by measuring the performance of several representative task-based and message-based process architectures in a controlled environment. Furthermore, our experiments report the impact of both context switching and synchronization overhead on communication subsystem performance. In addition to measuring data link, network, and transport layer performance, our experiments also measure presentation layer performance. The presentation layer is widely considered to be a major bottleneck in high-performance communication subsystems [CT90].

4.3 Multi-processor Platform

All experiments were conducted on an otherwise idle Sun SPARCcenter 2000 shared memory symmetric multi-processor. This SPARCcenter platform contained 640 Mbytes of RAM and 20 superscalar SPARC 40 MHz processing elements (PEs), each rated at approximately 135 MIPs. The operating system used for the experiments was release 5.3 of SunOS. SunOS 5.3 provides a multi-threaded kernel that allows multiple system calls and device interrupts to execute in parallel on the SPARCcenter platform [EKB+92]. All the process architectures in the experiments execute protocol tasks using separate SunOS unbound threads. These unbound threads are multiplexed over $1, 2, 3, \ldots 20$ SunOS lightweight processes (LWPs) within an OS process. The SunOS scheduler maps each LWP directly onto a separate kernel thread. Since kernel threads are the units of PE scheduling and execution in SunOS, multiple LWPs run protocol tasks in parallel on the SPARCcenter 20 PEs.

The memory bandwidth of the SPARCcenter is approximately 750 Mbits/sec. In addition to memory bandwidth, communication subsystem throughput is significantly affected by the context switching and synchronization overhead of the multi-processor platform. Scheduling and synchronizing a SunOS LWP requires a kernel-level context switch. This context switch flushes register windows and updates instruction and data caches, instruction pipelines, and translation lookaside buffers [MB91]. These activities take approximately 50 μsecs to perform between LWPs running in the same process. During this time, the PE incurring the context switch does not execute any protocol tasks.

ASX `Mutex` and `Condition` objects were both used in the experiments. `Mutex` objects were implemented using SunOS adaptive spin-locks and `Condition` objects were implemented using SunOS sleep-locks [EKB+92]. Synchronization methods invoked on `Condition` objects were approximately two orders of magnitude more expensive compared with methods on `Mutex` objects. For instance, measurements indicated that approximately 4 μsecs were required to acquire or release a `Mutex` object when no other PEs contended for the lock. In contrast, when all 20 PEs contended for a `Mutex` object, the time required to perform the locking methods increased to approximately 55 μsecs.

Approximately 300 μsecs were required to synchronize LWPs using `Condition` objects when no other PEs contended for the lock. Conversely, when all 20 PEs contended for a `Condition` object, the time required to perform the locking methods increased to approximately 520 μsecs. The two orders of magnitude difference in performance between `Mutex` and `Condition` objects was caused by the more complex algorithms used to implement `Condition` object methods. In addition, performing the `wait` method on a `Condition` object incurred a context switch, which further increased the synchronization overhead. In contrast, performing an `acquire` method on a `Mutex` object implemented with an adaptive spin-lock rarely triggered a context switch.

4.4 Functionality of the Communication Protocol Stacks

Two types of protocol stacks were investigated in the experiments. One was based on the connectionless UDP transport protocol; the other was based on the connection-oriented TCP transport protocol. Both protocol stacks contained data-link, network, transport, and presentation layers. The presentation layer was included in the experiments since it represents a major bottleneck in high-performance communication subsystems [CT90, GNI92].

Both the connectionless and connection-oriented protocol stacks were developed by specializing reusable components in the `ASX` framework via inheritance and parameterized types. As discussed in Section 3.3.4.1, inheritance and parameterized types were used to hold protocol stack functionality constant, while the process architecture was systematically varied. Each layer in a protocol stack was implemented as a `Module`, whose read-side and write-side inherit interfaces and implementations from the `Queue` abstract class described in Section 3.3.1. The synchronization and demultiplexing mechanisms required to implement different process architectures were parameterized using C++ template class arguments. As illustrated in Section 3.3.4.1, these templates were instantiated based upon the type of process architecture being tested.

The data-link layer in each protocol stack was implemented by the `DLP Module`. This `Module` transformed network packets received from a network interface into the canonical message format used internally by the interconnected `Queue` components in a Stream. Preliminary tests conducted with the widely-available `ttcp` benchmarking tool indicated that the SPARCcenter multi-processor platform processed messages through a protocol stack much faster than our 10 Mbps Ethernet network interface was capable of handling. Therefore, the network interface in our process architecture experiments was simulated with a single-copy pseudo-device driver operating in loop-back mode. This approach is consistent with those used in [Mat93, GNI92, NYKT94].

The network and transport layers of the protocol stacks were based on the IP, UDP, and TCP implementations in the BSD 4.3 Reno release [LMKQ89]. The 4.3 Reno TCP implementation contains the TCP header prediction enhancements, as well as the TCP slow start algorithm and congestion avoidance features [Jac88]. The UDP and TCP transport protocols were configured into the `ASX` framework via the `UDP` and `TCP Modules`. Network layer processing was performed by the `IP Module`. This `Module` handled routing and segmentation/reassembly of Internet Protocol (IP) packets.

Presentation layer functionality was implemented in the `XDR Module` using marshaling routines produced by the ONC eXternal Data Representation (XDR) stub generator (`rpcgen`) [Sun87]. The ONC XDR stub generator translates type specifications into marshaling routines. These marshaling routines encode/decode implicitly-typed messages before/after exchanging them among hosts that may possess heterogeneous processor byte-orders. The ONC presentation layer conversion mechanisms consist of a type specification language (XDR) and a set of library routines that implement the appropriate encoding and decoding rules for built-in integral types (*e.g.,* char, short, int, and long), as well as real types (*e.g.,* float and double). These library routines may be combined to produce marshaling routines for arbitrarily complex user-defined composite types (such as record/structures, unions, arrays, and pointers). Messages exchanged via XDR are implicitly-typed, which improves marshaling performance at the expense of run-time flexibility.

The `XDR` routines generated for the connectionless and connection-oriented protocol stacks converted incoming and outgoing messages into and from variable-sized arrays of structures containing a set of integral and real values. The `XDR` processing involved byte-order conversions, as well as dynamic memory allocation and deallocation.

4.5 Structure of the Process Architectures

The remainder of this section outlines the structure of the message-based and task-based process architectures used to parallelize the connectionless and connection-oriented protocol stacks described above.

4.5.1 Structure of the Message-based Process Architectures

4.5.1.1 Connectional Parallelism

The protocol stack depicted in Figure 4.1 illustrates an `ASX`-based implementation of the Connectional Parallelism (`CP`) process architecture outlined in Section 2.3.1.1. Each process performs the data-link, network, transport, and presentation layer tasks sequentially for a single connection. Protocol tasks are divided into four interconnected `Modules`, corresponding to the data-link, network, transport, and presentation layers.

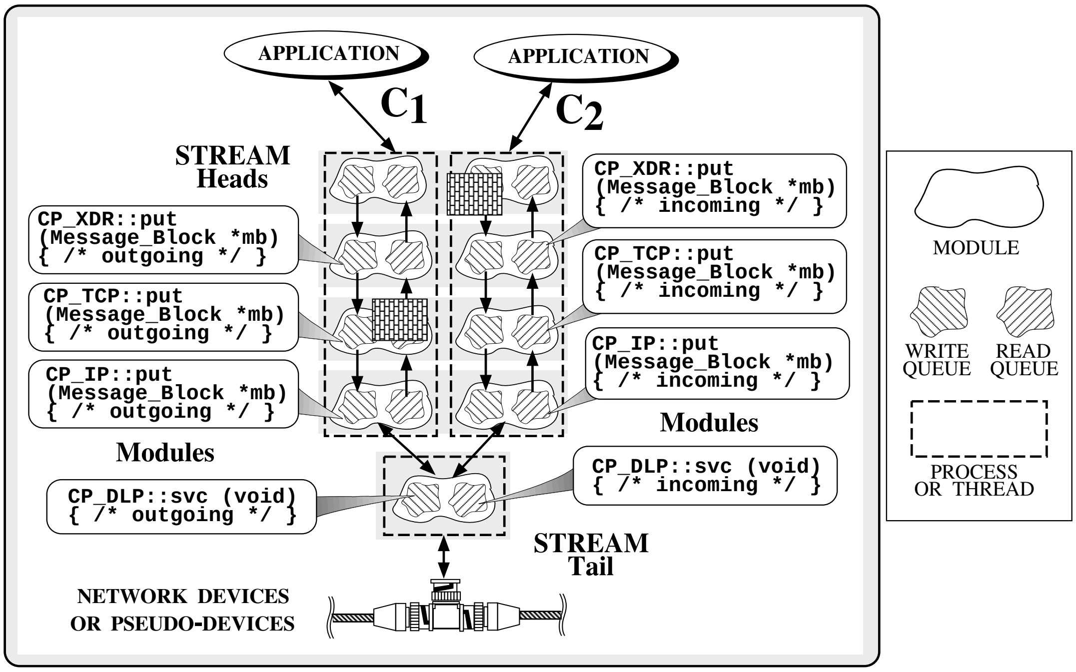

Figure 4.1: Connectional Parallelism Process Architecture

Data-link processing is performed in the `CP_DLP Module`. The Connectional Parallelism implementation of this `Module` performs "eager demultiplexing" via a packet filter at the data-link layer. Thus, the `CP_DLP Module` uses its read-side `svc` method to demultiplex incoming messages onto the appropriate transport layer connection. In contrast, the `CP_IP`, `CP_TCP`, and `CP_XDR Modules` perform their processing synchronously in their respective `put` methods. To eliminate extraneous data movement overhead, the `Queue::put_next` method passes a pointer to a message between protocol layers.

4.5.1.2 Message Parallelism

Figure 4.2 depicts the Message Parallelism (`MP`) process architecture used for the TCP-based connection-oriented protocol stack. When an incoming message arrives, it is handled by the `MP_DLP::svc` method. This method manages a pool of pre-spawned

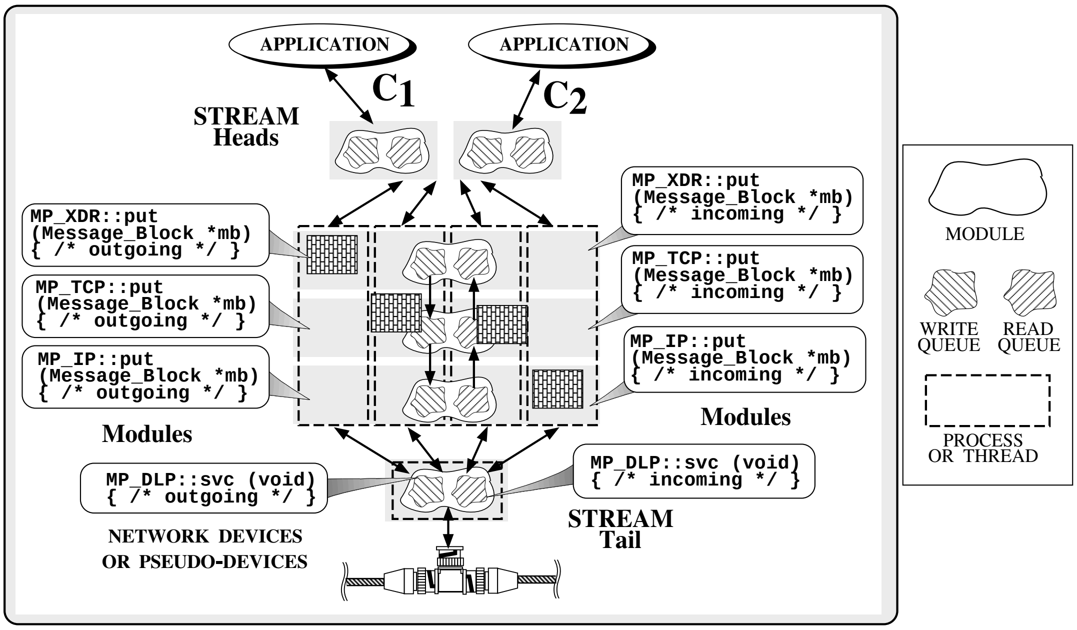

Figure 4.2: Message Parallelism Process Architecture

SunOS unbound threads. Each message is associated with an unbound thread that escorts the message synchronously through a series of interconnected `Queues` that form a protocol stack. Each layer of the protocol stack performs the protocol tasks defined by its `Queue`. When these tasks are complete, an upcall [Cla85] may be used to pass the message to the next adjacent layer in the protocol stack. The upcall is performed by invoking the `put` method in the adjacent layer's `Queue`. This `put` method borrows the thread of control from its caller and executes the protocol tasks associated with its layer.

The Message Parallelism process architecture for the connectionless protocol stack is similar to the one used to implement the connection-oriented protocol stack. The primary difference between the two protocol stacks is that the connectionless stack performs UDP transport functionality, which is less complex than TCP. For example, UDP does not generate acknowledgements, keep track of round-trip time estimates, or manage congestion windows. In addition, the connectionless `MP_UDP::put` method handles each message concurrently and independently, without explicitly preserving inter-message ordering. In contrast, the connection-oriented `MP_TCP::put` method utilizes several `Mutex` synchronization objects. As separate messages from the same connection ascend the protocol stack in parallel, these `Mutex` objects serialize access to per-connection control blocks. Serialization is required to protect shared resources (such as message queues, protocol connection records, TCP segment reassembly, and demultiplexing tables) against race conditions.

Both Connectional Parallelism and Message Parallelism optimize message management by using SunOS thread-specific storage [EKB+92] to buffer messages as they flow through a protocol stack. This optimization leverages off the cache affinity properties of the SunOS shared memory multi-processor. In addition, it minimizes the cost of synchronization operations used to manage the global dynamic memory heap.

4.5.2 Structure of the Task-based Process Architecture

4.5.2.1 Layer Parallelism

Figure 4.3 illustrates the `ASX` framework components that implement a Layer Parallelism (`LP`) process architecture for the TCP-based connection-oriented protocol stack. The connectionless UDP-based protocol stack for Layer Parallelism was designed in a similar manner. The primary difference between them was that the read-side and write-side `Queues` in the connectionless transport layer `Module` (`LP_UDP`) implement the simpler UDP functionality.

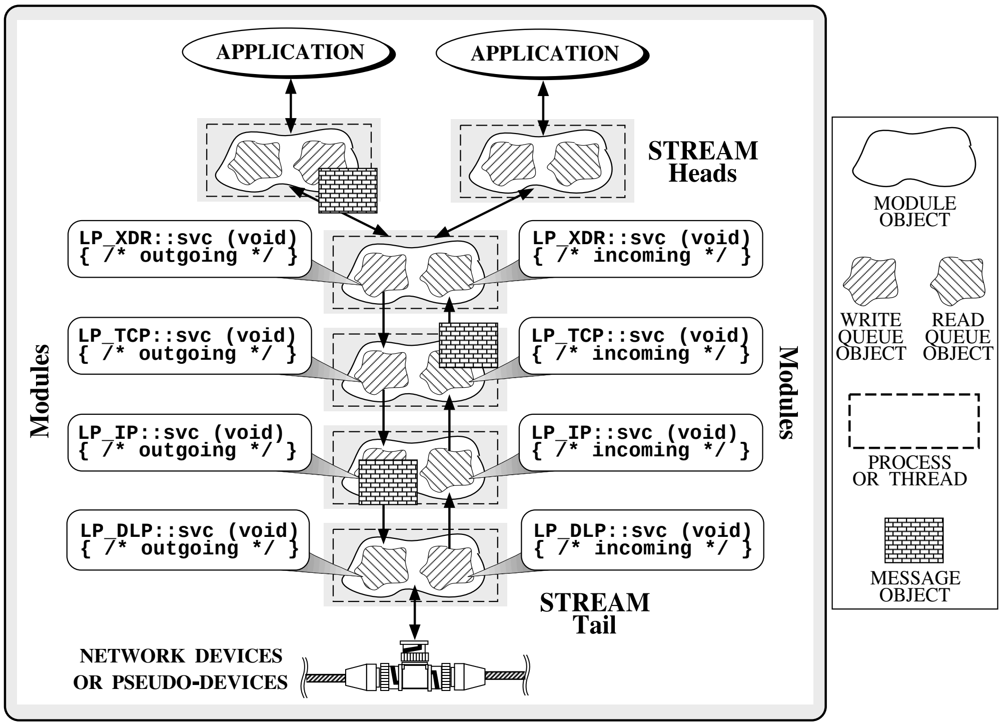

Figure 4.3: Layer Parallelism Process Architecture

Protocol-specific processing at each protocol layer shown in Figure 4.3 is performed in the `Queue::svc` method. Each `svc` method is executed in a separate process associated with the `Module` that implements the corresponding protocol layer (*e.g.,* `LP_XDR`, `LP_TCP`, `LP_IP`, and `LP_DLP`). These processes cooperate in a producer/consumer manner, operating in parallel on message header and data fields corresponding to their particular protocol layer. Every `svc` method performs its protocol layer tasks before passing a message to an adjacent `Module` running in a separate process.

All processes share a common address space, which eliminates the need to copy messages that are passed between adjacent `Modules`. However, even if pointers to messages are passed between processes, per-PE data caches may be invalidated by hardware cache consistency protocols. Cache invalidation degrades performance by increasing the level of contention on the SPARCcenter system bus. Moreover, since messages are passed between PEs, the memory management optimization techniques (described in Section 4.5.1) that used the thread-specific storage are not applicable.

A strict implementation of Layer Parallelism would limit parallel processing to only include the number of protocol layers that could run on separate PEs. On a platform with 2 to 4 PEs this is not a serious problem since the protocol stacks used in the experiments only had 4 layers. On the 20 PE SPARCcenter platform, however, this approach would have greatly constrained the ability of Layer Parallelism to utilize the available processing resources. To alleviate this constraint, the connection-oriented Layer Parallelism process architecture was implemented to handle a cluster of connections (*i.e.,* 5 connections per 4 layer protocol stack, with one PE per-layer). Likewise, the connectionless Layer Parallelism process architecture was partitioned across 5 network interfaces to utilize the available parallelism.

4.6 Measurement Results

This section presents results obtained by measuring the data reception portion of the protocol stacks developed using the process architectures described in Section 4.5. Three types of measurements were obtained for each combination of process architecture and protocol stack: *average throughput*, *context switching overhead*, and *synchronization overhead*. Average throughput measured the impact of parallelism on protocol stack performance. Context switching and synchronization measurements were obtained to help explain the variation in the average throughput measurements.

Average throughput was measured by holding the protocol functionality, application traffic, and network interfaces constant, while systematically varying the process architecture in order to determine the impact on performance. Each benchmarking run measured the amount of time required to process 20,000 4 kbyte messages. In addition, 10,000 4 kbyte messages were transmitted through the protocol stacks at the start of each run to ensure that all the PE caches were fully initialized (the time required to process these initial 10,000 messages was not used to calculate the throughput performance). Each test was run using $1, 2, 3, \ldots 20$ PEs, with each test replicated a dozen times and the results averaged. The purpose of replicating the tests was to insure that the amount of interference from internal OS process management tasks did not perturb the results.

Various statistics were collected using an extended version of the widely available `ttcp` protocol benchmarking tool [USN84]. The `ttcp` tool measures the amount of OS processing resources, user-time, and system-time required to transfer data between a transmitter process and a receiver process. The flow of data is uni-directional, with the transmitter flooding the receiver with a user-specified number of data buffers. Various sender and receiver parameters (such as the number of data buffers transmitted and the size of data buffers and protocol windows) may be selected at run-time.

The version of `ttcp` used in our experiments was modified to use `ASX`-based connection-oriented and connectionless protocol stacks. These protocol stacks were configured in accordance with the process architectures described in Section 4.5. The `ttcp` tool was also enhanced to allow a user-specified number of connections to be active simultaneously. This extension enabled us to measure the impact of multiple connections on the performance of the connection-oriented protocol stacks using message-based and task-based process architectures.

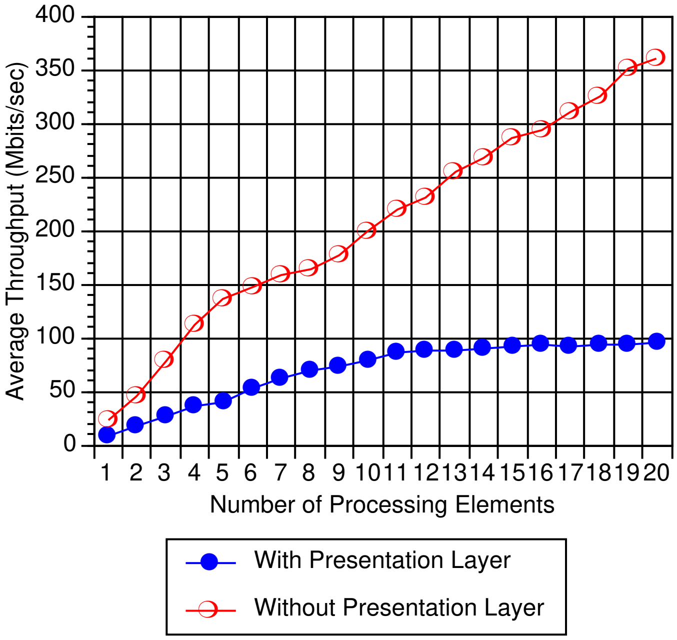

Figure 4.4: Connection-oriented Connectional Parallelism Throughput

4.6.1 Throughput Measurements

Figures 4.4, 4.5, and 4.6 depict the average throughput for the message-based process architectures (Connectional Parallelism and Message Parallelism) and the task-based process architecture (Layer Parallelism) used to implement the connection-oriented protocol stacks. Each test run for these connection-oriented process architectures used 20 connections. These figures report the average throughput (in Mbits/sec), measured both with and without presentation layer processing. The figures illustrate how throughput is affected as the number of PEs increase from 1 to 20. Figures 4.7, 4.8, and 4.9 indicate the relative speedup that resulted from successively adding another PE to each process architecture. Relative speedup is computed by dividing the average aggregated throughput for n PEs (shown in Figures 4.4, 4.5, and 4.6, where $1 \leq n \leq 20$) by the average throughput for 1 PE.

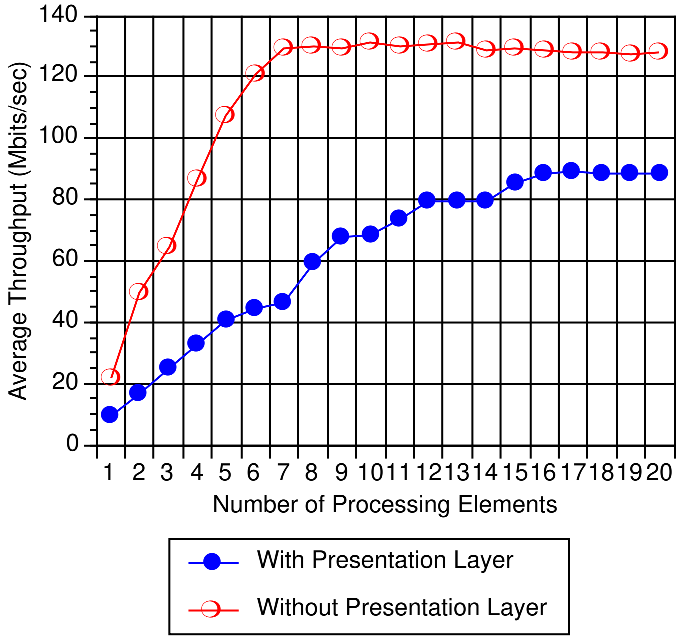

Figure 4.5: Connection-oriented Message Parallelism Throughput

The results from Figures 4.4, 4.5, 4.7, and 4.8 indicate that increasing the number of PEs generally improves the average throughput in the message-based process architectures. Connection-oriented Connectional Parallelism exhibited the highest performance, both in terms of average throughput and in terms of relative speedup. As shown in Figure 4.4, the average throughput of Connectional parallelism with presentation layer processing peaks at approximately 100 Mbits/sec. The average throughput without presentation layer processing peaks at just under 370 Mbits/sec. These results indicate that the presentation layer represents a significant portion of the overall protocol stack overhead. As shown in Figure 4.7, the relative speedup of Connectional Parallelism without presentation layer processing increases steadily from 1 to 20 PEs. The relative speedup with presentation layer processing is similar up to 12 PEs, at which point it begins to level off. This speedup curve flattens due to the additional overhead from data movement and synchronization performed in the presentation layer.

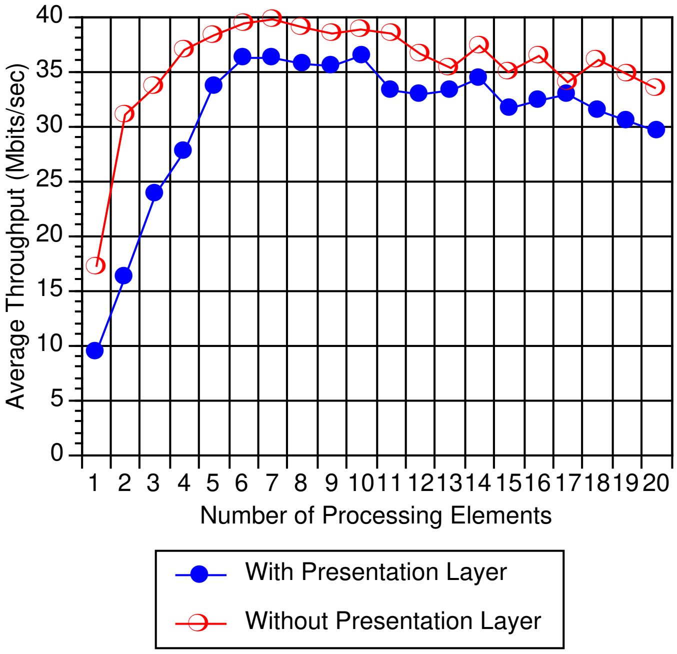

Figure 4.6: Connection-oriented Layer Parallelism Throughput

As shown in Figure 4.5, the average throughput achieved by connection-oriented Message Parallelism without presentation layer processing peaks at 130 Mbits/sec. When presentation layer processing is performed, the average throughput is 1.5 to 3 times lower, peaking at approximately 90 Mbits/sec. Note, however, that the relative speedup without presentation layer processing (shown in Figure 4.8) flattens out after 8 CPUs. This speedup curve flattens out when presentation layer processing is omitted due to increased contention for shared synchronization objects at the transport layer. This synchronization overhead is discussed further in Section 4.6.3. In contrast, the relative speedup of connection-oriented Message Parallelism with presentation layer processing grows steadily from 1 to 20 PEs. This behavior suggests that connection-oriented Message Parallelism benefits more from parallelism when the protocol stack contains presentation layer processing. This finding is consistent with those reported in [NYKT94], and

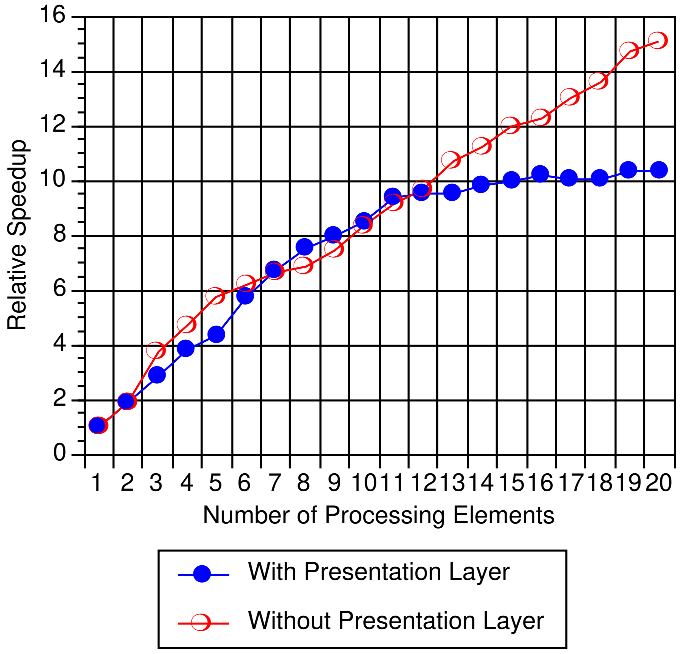

Figure 4.7: Relative Speedup for Connection-oriented Connectional Parallelism

is related to the minimal synchronization requirements of the presentation layer (compared with the higher relative amounts of synchronization in the connection-oriented transport layer).

In contrast to Connectional Parallelism and Message Parallelism, the performance of the connection-oriented Layer Parallelism (shown in Figures 4.6 and 4.9) did not scale up as the number of PEs increased. The average throughput with presentation layer processing (shown in Figure 4.6) peaks at approximately 36 Mbits/sec. This amount is much less than half the throughput achieved by Connectional Parallelism and Message Parallelism. The throughput exhibited by Layer Parallelism peaks at 40 Mbits/sec when presentation layer processing is omitted. This is over 3 times lower than Message Parallelism and approximately 9 times lower than Connectional Parallelism. As shown in Figure 4.9, the relative speedup both with and without presentation layer processing increases until after 10 and 7 PEs, respectively. After peaking, average throughput levels off and gradually begins to decrease. This decrease in Layer Parallelism performance

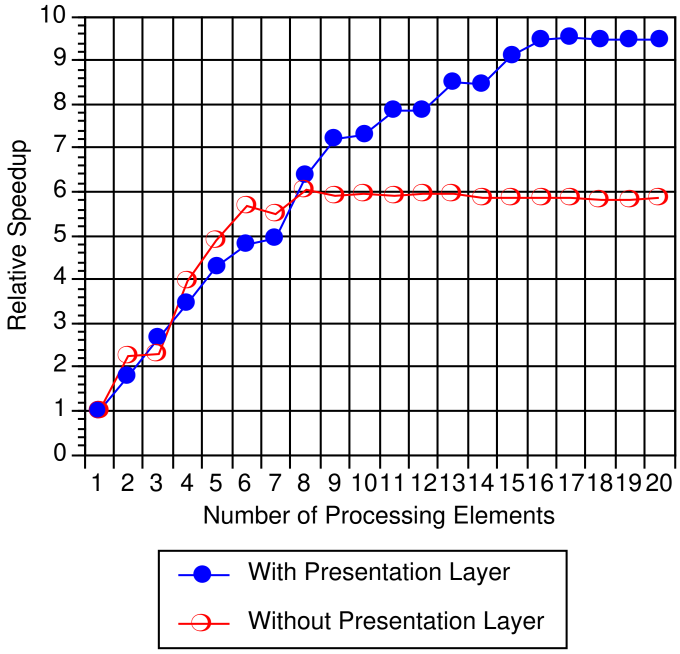

Figure 4.8: Relative Speedup for Connection-oriented Message Parallelism

occurs from the high levels of context switching (discussed in Section 4.6.2), as well as the high levels of `Condition` object synchronization overhead (discussed in Section 4.6.3).

A limitation with Connectional Parallelism is that each individual connection executes sequentially. Therefore, Connectional Parallelism becomes most effective as the number of connections approaches the number of PEs. In contrast, Message Parallelism utilizes multiple PEs more effectively when the number of connections is much less than the number of PEs. Figure 4.10 illustrates this point by graphing average throughput as a function of the number of connections. This test held the number of PEs constant at 20, while increasing the number of connections from 1 to 20. Connectional Parallelism consistently out-performs Message Parallelism as the number of connections becomes greater than 10.

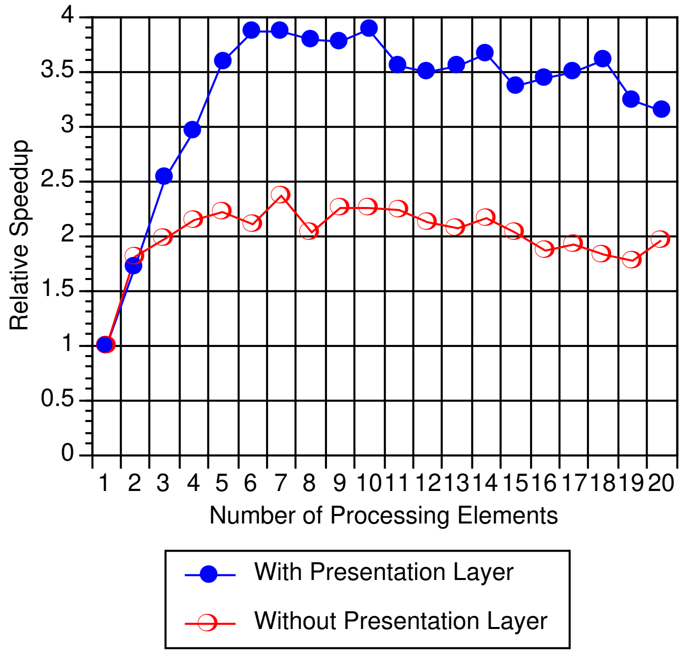

Figure 4.9: Relative Speedup for Connection-oriented Layer Parallelism

Figures 4.11 and 4.12 depict the average throughput for the message-based and task-based process architectures used to implement the connectionless protocol stacks (note that Connectional Parallelism is not applicable for a connectionless protocol stack). These figures report the throughput measured both with and without presentation layer processing. Figures 4.13 and 4.14 indicate the speedup of each process architecture, relative to its single PE case, for the data points reported in Figures 4.11 and 4.12.

The connectionless message-based process architecture (Figure 4.11) significantly outperforms the task-based process architecture (Figure 4.12). This behavior is consistent with the results from the connection-oriented tests shown in Figure 4.4 through Figure 4.9. With presentation layer processing, the throughput and relative speedup of connectionless Message Parallelism is slightly higher than the connection-oriented version shown in Figures 4.5 and 4.8. However, without presentation layer processing, the throughput and relative speedup of connectionless Message Parallelism are substantially higher (500 Mbits/sec vs. 130 Mbits/sec). In addition, note that the relative speedup of

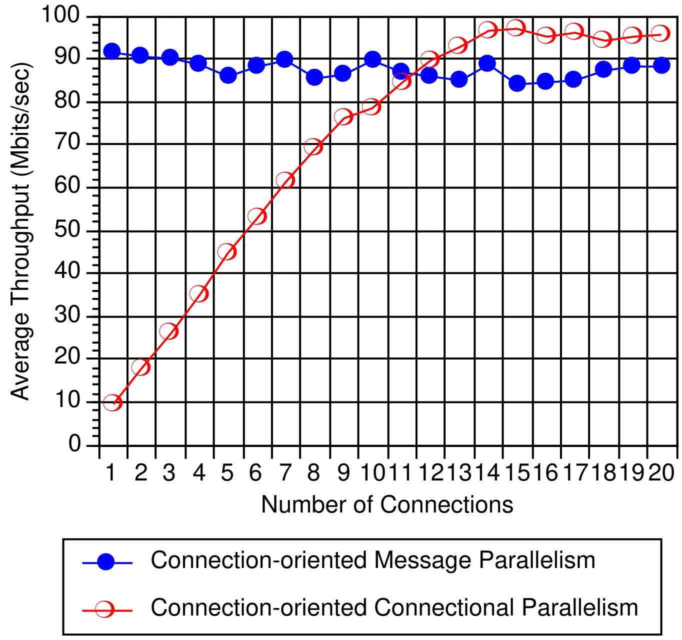

Figure 4.10: Comparison of Connectional Parallelism and Message Parallelism

connectionless Message Parallelism without presentation layer processing exceeds that attained with presentation layer processing (shown in Figure 4.13). This behavior is the reverse of the connection-oriented Message Parallelism results (shown in Figure 4.8). The difference in performance is due to the fact that connectionless Message Parallelism incurs much lower levels of synchronization overhead (synchronization is discussed further in Section 4.6.3).

The throughput of the connectionless Layer Parallelism (shown in Figure 4.12) suffered from the same problems as the connection-oriented version (shown in Figure 4.6). As shown in Figure 4.9, the relative speedup increases only up to 6 PEs, regardless of whether presentation layer processing was performed or not. At this point, the performance levels off and begins to decrease. This behavior is accounted for by the high levels of Layer Parallelism context switching discussed in the following section.

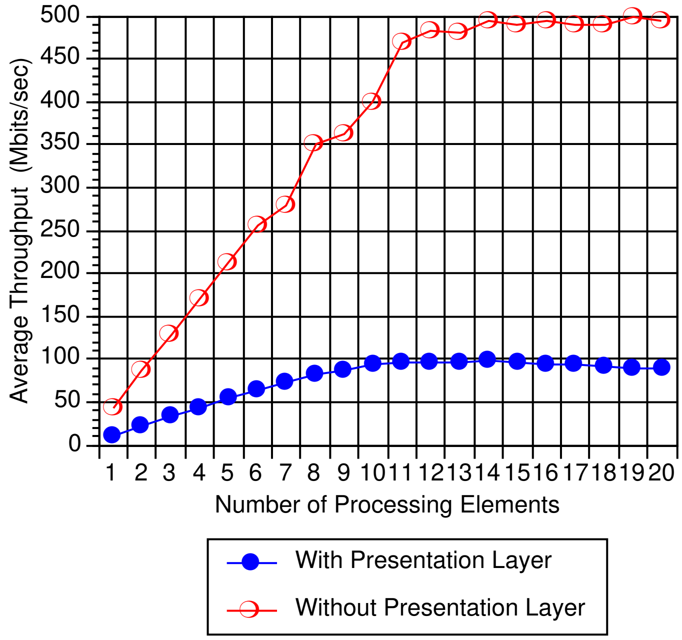

Figure 4.11: Connectionless Message Parallelism Throughput

4.6.2 Context Switching Measurements

Measurements of context switching overhead were obtained by modifying the `ttcp` benchmarking tool to use the SunOS 5.3 `/proc` process file system. The `/proc` file system provides access to the executing image of each process in the system. It reports the number of voluntary and involuntary context switches incurred by SunOS LWPs within a process. Figures 4.15 through 4.19 illustrate the number of voluntary and involuntary context switches incurred by transmitting the 20,000 4 kbyte messages through the process architectures and protocol stacks measured in this study.

A voluntary context switch is triggered when a protocol task puts itself to sleep awaiting certain resources (such as I/O devices or synchronization locks) to become available. For example, a protocol task may attempt to acquire a resource that is not available immediately (such as obtaining a message from an empty list of messages in a `Queue`). In this case, the protocol task puts itself to sleep by invoking the `wait` method

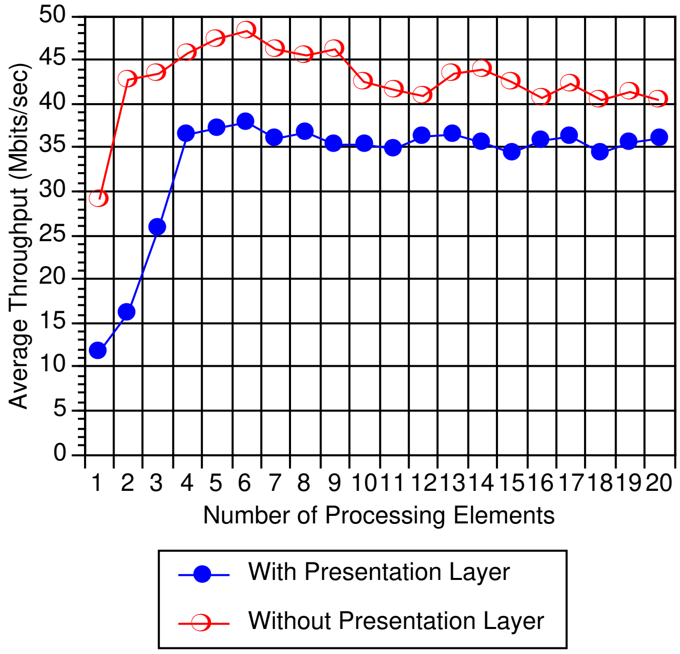

Figure 4.12: Connectionless Layer Parallelism Throughput

of an `ASX Condition` object. This method causes the SunOS kernel to preempt the current thread of control and perform a context switch to another thread of control that is capable of executing protocol tasks immediately. For each combination of process architecture and protocol stack, voluntary context switching increases fairly steadily as the number of PEs increase from 1 through 20 (shown in Figures 4.15 through 4.19).

An involuntary context switch occurs when the SunOS kernel preempts a running unbound thread in order to schedule another thread of control to execute other protocol tasks. The SunOS scheduler preempts an active thread of control every 10 milliseconds when the time-slice alloted to its LWP expires. Note that the rate of growth for involuntary context switching shown in Figures 4.15 through 4.19 remains fairly consistent as the number of PEs increase. Therefore, it appears that most of the variance in average throughput performance is accounted for by voluntary context switching, rather than by involuntary context switching.

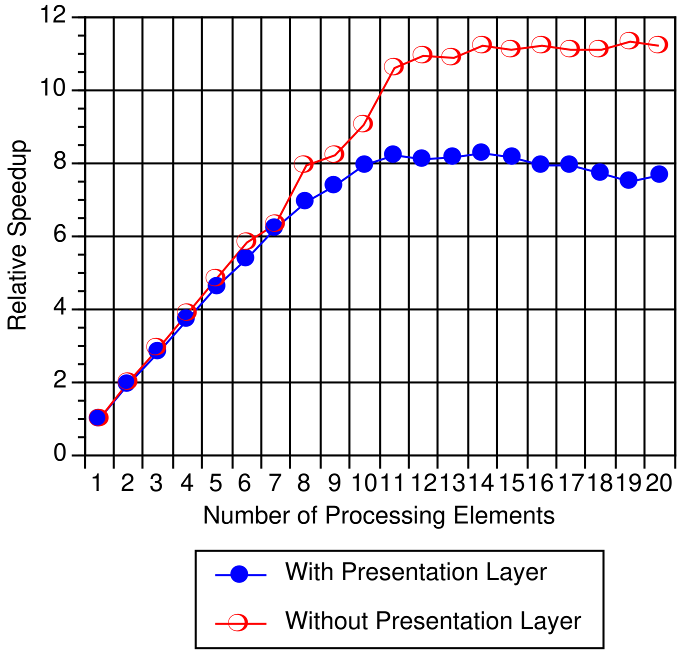

Figure 4.13: Relative Speedup for Connectionless Message Parallelism

The task-based process architectures (shown in Figures 4.18 and 4.19) exhibited approximately 4 to 5 times higher levels of voluntary context switching than the message-based process architectures (shown in Figures 4.15, 4.16, and 4.17). This difference stems from the synchronization mechanisms used for the Layer Parallelism process architecture. This process architecture uses sleep-locks to implement flow control between protocol stack layers running in separate PEs. This type of flow control is necessary since processing activities in each layer may execute at different rates. In SunOS, the `wait` and `signal` methods on `Condition` objects are implemented using sleep-locks, which trigger voluntary context switches. In contrast, Connectional Parallelism and Message Parallelism use adaptive spin-lock synchronization, which is less costly since it typically does *not* trigger voluntary context switches. The substantially lower-levels of voluntary context switching exhibited by Connectional Parallelism and Message Parallelism helps to account for their consistently higher overall throughput and greater relative speedup discussed in Section 4.6.1.

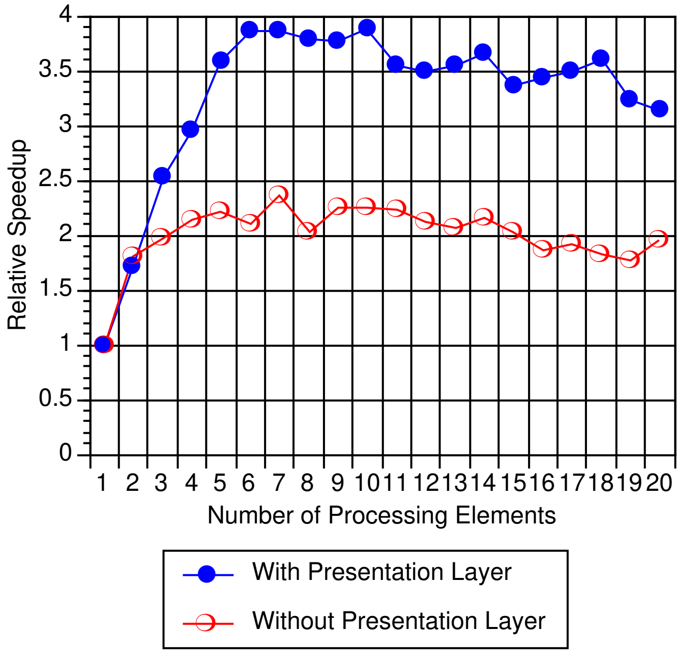

Figure 4.14: Relative Speedup of Connectionless Layer Parallelism

As shown in Figure 4.15, Connectional Parallelism incurred the lowest levels of context switching for the connection-oriented protocol stacks. In this process architecture, after a message has been demultiplexed onto a connection, all that connection's context information is directly accessible within the address space of the associated thread of control. In general, a thread of control in Connectional Parallelism processes its connection's messages without incurring additional context switching overhead.

4.6.3 Synchronization Measurements

Measurements of synchronization overhead were collected to determine the amount of time spent acquiring and releasing locks on `ASX` `Mutex` and `Condition` objects during protocol processing on the 20,000 4 kbyte messages. Unlike context switches, the SunOS 5.3 `/proc` file system does not maintain accurate metrics on synchronization overhead. Therefore, these measurements were obtained by bracketing the `Mutex`

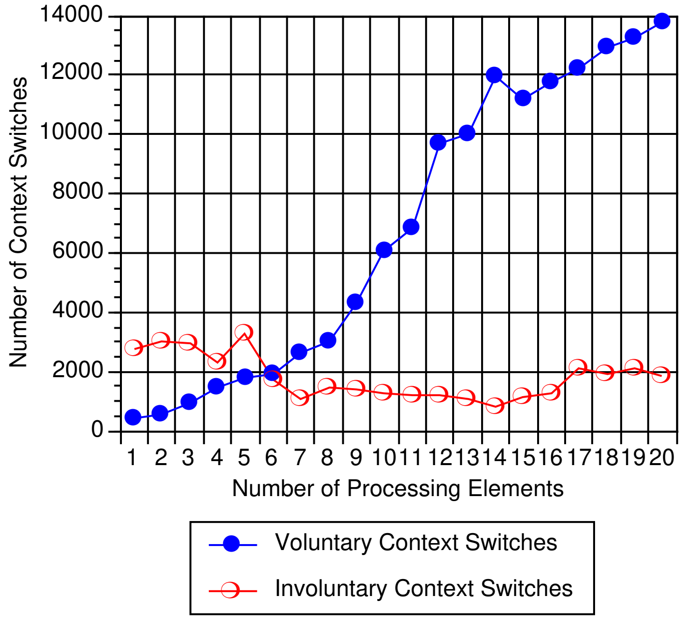

Figure 4.15: Connection-oriented Connectional Parallelism Context Switching

and `Condition` methods with calls to the `gethrtime` system call. This system call uses the SunOS 5.3 high-resolution timer, which expresses time in nanoseconds from an arbitrary time in the past. The time returned by the `gethrtime` system call is not subject to resetting or drifting since it is not correlated with the current time of day.

Figures 4.20 through 4.24 indicate the total time (measured in msecs) used to acquire and release locks on `Mutex` and `Condition` synchronization objects. These tests were performed using all three process architectures to implement connection-oriented and connectionless protocol stacks that contained data-link, network, transport, and presentation layer functionality. The message-based process architectures (Connectional Parallelism and Message Parallelism, shown in Figures 4.20, 4.21, and 4.23) used `Mutex` synchronization mechanisms that utilize adaptive spin-locks (which rarely trigger a context switch). In contrast, the task-based process architecture (Layer Parallelism, shown in Figures 4.22 and 4.24) utilized both `Mutex` and `Condition` objects (`Condition` objects *do* trigger context switches).

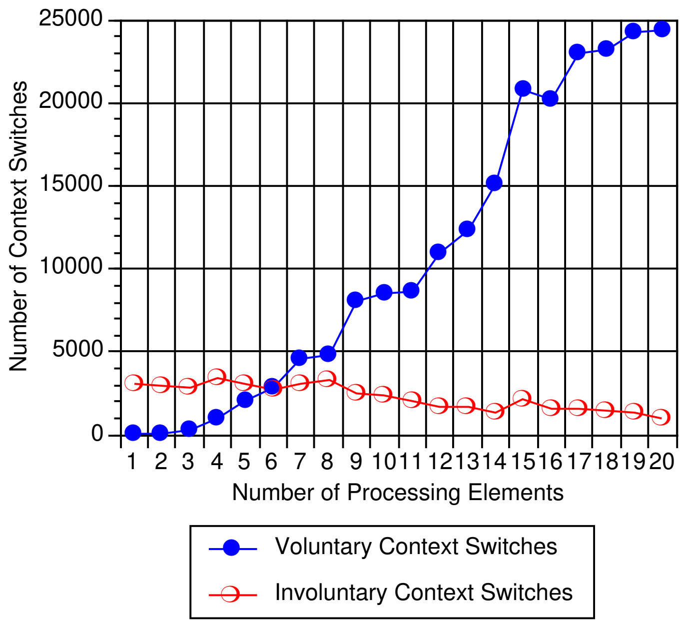

Figure 4.16: Connection-oriented Message Parallelism Context Switching

Connection-oriented Connectional Parallelism (shown in Figure 4.20) exhibited the lowest levels of synchronization overhead, which peaked at approximately 700 msecs. This synchronization overhead was approximately 1 order of magnitude lower than the results shown in Figures 4.21 through 4.24. Moreover, the amount of synchronization overhead incurred by Connectional Parallelism did not increase significantly as the number of PEs increased from 1 to 20. This behavior occurs since after a message is demultiplexed onto a PE/connection, few additional synchronization operations are required. In addition, since Connectional Parallelism processes messages within a single PE cache, it leverages off of SPARCcenter 2000 multi-processor cache affinity properties [VZ91].

The synchronization overhead incurred by connection-oriented Message Parallelism (shown in Figure 4.21) peaked at just over 6,000 msecs. Moreover, the rate of growth increased fairly steadily as the number of PEs increased from 1 to 20. This behavior occurs from the lock contention caused by `Mutex` objects that serialize access

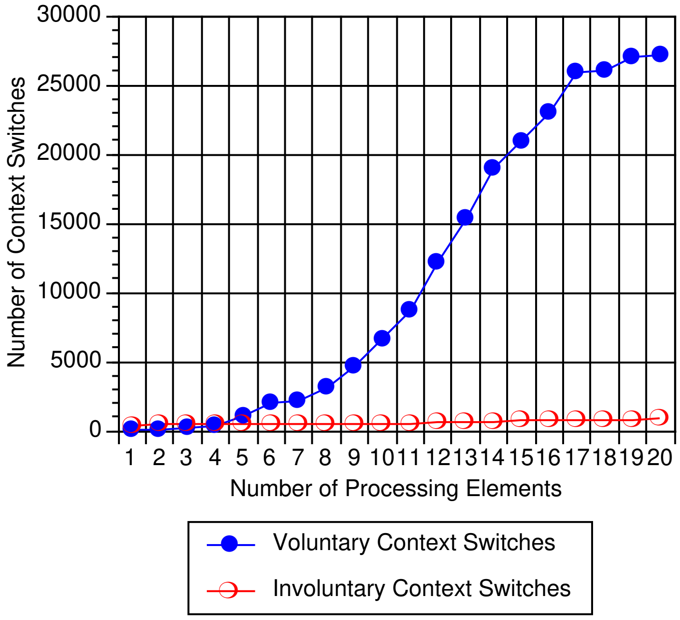

Figure 4.17: Connectionless Message Parallelism Context Switching

to the `Map_Manager` connection demultiplexer (discussed in Section 3.3.4.1). In contrast, the connectionless Message Parallelism protocol stack does not require the use of this connection demultiplexer. Therefore, the amount of synchronization overhead it incurred was much lower, peaking at under 1,800 msecs.

Connection-oriented Layer Parallelism exhibited two types of synchronization overhead (shown in Figure 4.22). The amount of overhead resulting from `Mutex` objects peaked at just over 2,000 msecs, which was lower than that of connection-oriented Message Parallelism (shown in Figure 4.21). However, the amount of synchronization overhead from the `Condition` objects was much higher, peaking at approximately 18,000 msecs (shown in Figure 4.22). In the Layer Parallelism implementation, the `Condition` objects implemented flow control between separate layers executing on different PEs in a protocol stack. The connectionless version of Layer Parallelism also exhibited high levels of synchronization overhead (shown in Figure 4.24).

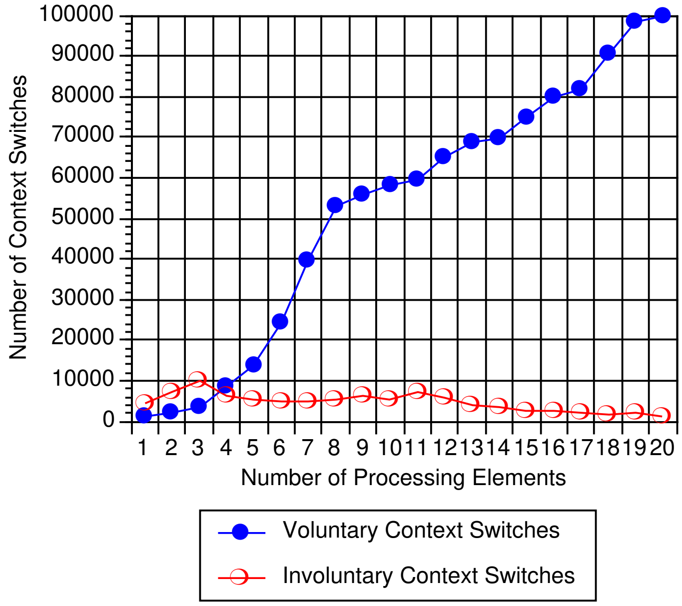

Figure 4.18: Connection-oriented Layer Parallelism Context Switching

4.7 Summary

The following observations resulted from our experience gained by conducting performance experiments on alternative process architectures for connectionless and connection-oriented protocol stacks:

- Implementing the task-based process architectures was relatively straightforward. These process architectures map onto conventional layered communication models using well-structured "producer/consumer" designs [Atk88]. Minimal synchronization was necessary *within* a layer since parallel processing was serialized at a service access point (such as the service access point defined between the network and transport layers). However, as shown by the performance experiments, the task-based Layer Parallelism process architecture exhibited high levels of context switching and synchronization overhead on the SunOS shared memory multi-processor platform.

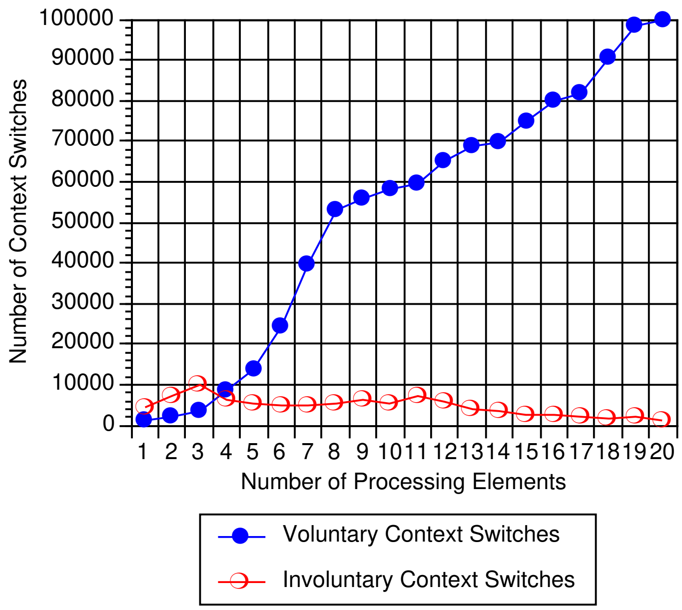

Figure 4.19: Connectionless Layer Parallelism Context Switching

- Implementing the message-based process architectures was challenging since the concurrency control mechanisms were more complex. However, the message-based process architectures used parallelism more effectively than the task-based process architectures. This is due in part to the fact that message-based process architecture parallelism was based upon dynamic characteristics (such as messages or connections). This dynamism enables the message-based process architectures to use a larger number of PEs effectively. As described in Section 4.6.1, the relative speedups gained from parallel message-based process architectures scaled up to use a relatively high number of PEs. In contrast, the parallelism used by the task-based process architectures depended on relatively static characteristics (such as the number of layers or protocol tasks), which did not scale up. In addition, the higher rate of growth for context switching and synchronization (discussed in Sections 4.6.2 and 4.6.3) hampered the ability of Layer Parallelism to effectively utilize a large number of PEs.

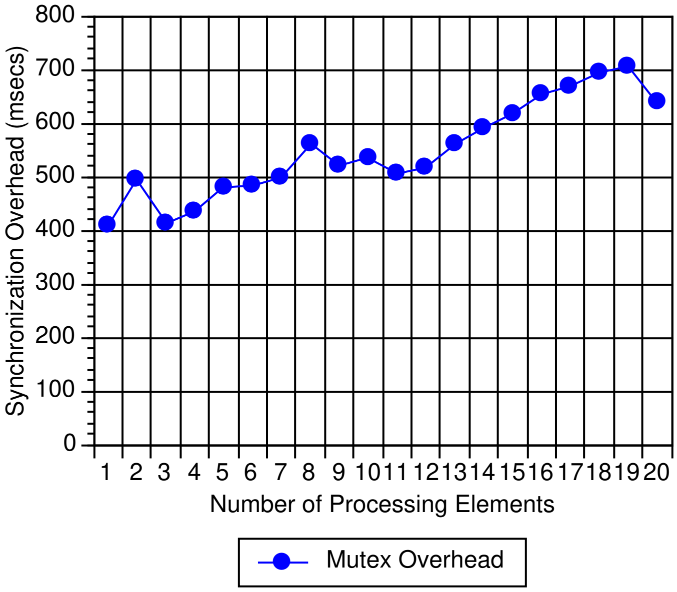

Figure 4.20: Connection-oriented Connectional Parallelism Synchronization Overhead

- Connectional Parallelism becomes more suitable than Message Parallelism as the number of connections approaches the number of PEs. Message Parallelism, on the other hand, is more suitable when the number of active connections is significantly less than the number of available PEs. In addition, unlike Connectional Parallelism, Message Parallelism is suitable for connectionless applications.

- It appears that connection-oriented Message Parallelism benefits more from parallelism when the protocol stack contains presentation layer processing. As shown in Figure 4.8, the speedup curve for connection-oriented Message Parallelism without presentation layer processing flattens out after 8 PEs. In contrast, when presentation layer processing is performed, the speedup continues until 16 PEs. This behavior results from the relatively low amount of synchronization overhead associated with parallel processing at the presentation layer. In contrast, the relative speedup for Connectional Parallelism without presentation layer processing continues to increase steadily up to 20 PEs (shown in Figure 4.7). Connectional

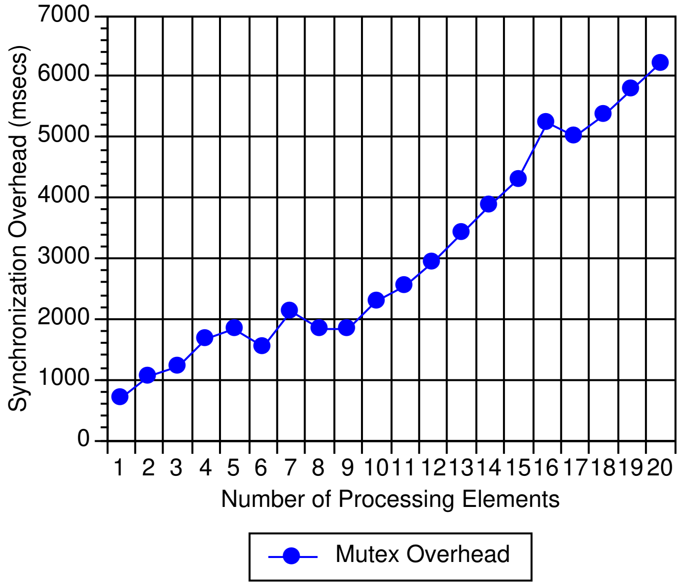

Figure 4.21: Connection-oriented Message Parallelism Synchronization Overhead

Parallelism performs well in this case due to its low levels of synchronization and context switching overhead.

- It appears that the relative cost of synchronization operations has a substantial impact on process architecture performance. On the SPARCcenter 2000 shared memory multi-processor running SunOS 5.3, the message-based process architectures benefit from their use of inexpensive adaptive spin-locks. In contrast, the task-based process architectures were penalized by the much higher (*i.e.,* two orders of magnitude) cost of sleep-lock synchronization. We conjecture that a multi-processor platform possessing different synchronization properties would produce significantly different results. For example, if the experiments reported in this chapter were replicated on a non-shared memory, message-passing transputer platform [Zit91], it is likely that the performance of the task-based process architectures would improve relative to the message-based process architectures.

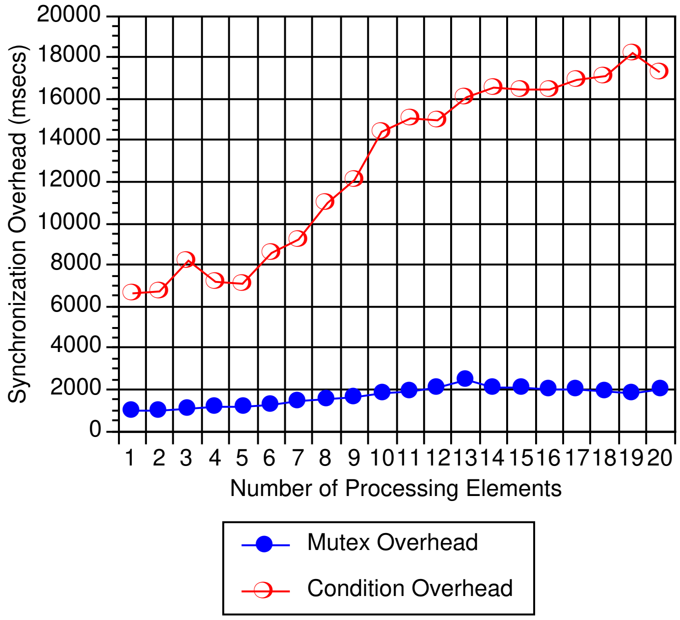

Figure 4.22: Connection-oriented Layer Parallelism Synchronization Overhead

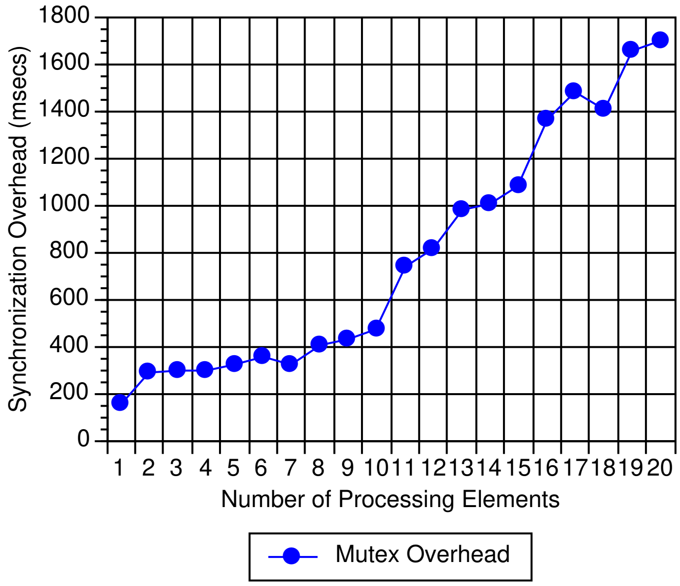

Figure 4.23: Connectionless Message Parallelism Synchronization Overhead

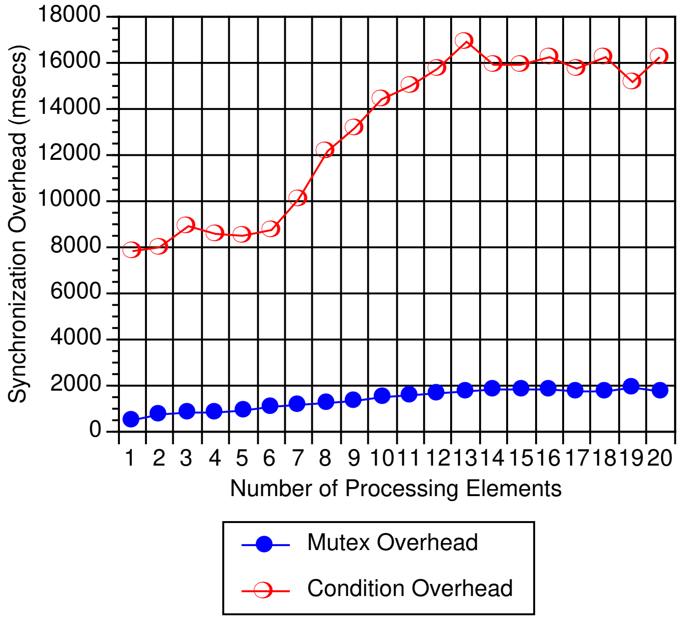

Figure 4.24: Connectionless Layer Parallelism Synchronization Overhead

Chapter 5

Conclusions and Future Research Problems

This paper describes performance measurements obtained by using the ASX framework to parallelize a connection-oriented protocol stack implemented with Connectional Parallelism and Message Parallelism process architectures. The ASX framework provides an integrated set of object-oriented components that facilitate experimentation with different types of process architectures on multi-processor platforms. By decoupling the protocol-specific functionality from the underlying process architecture, the ASX framework increased component reuse and simplified the development, configuration, and experimentation with parallel protocol stacks.

The experimental results presented in this dissertation demonstrate that to increase performance significantly, the speed-up obtained from parallelizing a protocol stack must outweight the context switching and synchronization overhead associated with parallel processing. If these sources of overhead are large, parallelizing a protocol stack will not yield substantial benefits. The task-based Layer Parallelism process architecture exhibited high levels of context switching and synchronization, and did not effectively utilize the available multi-processing resources on a SPARCcenter 2000 shared memory multi-processor platform containing 20 processing elements. In contrast, the message-based process architectures (Connectional Parallelism and Message Parallelism) incurred significantly less context switching and synchronization overhead, and exhibited much higher levels of performance and multi-processing resource utilization. In general, the results from these experiments underscore the importance of the process architecture on parallel communication subsystem performance.

Selecting an appropriate process architecture is an important design consideration in application domains other than communication subsystems. Over the next several years, I plan to replicate my performance experiments using more powerful multi-processor end-systems (based on both shared memory and message passing computer architectures – such as Transputers) and higher-capacity networks (such as ATM) in order to investigate the scalability of the alternative process architectures on a range of applications. For example, I am currently working on generalizing my dissertation research to address more general distributed system topics involving system and network management (such as high-speed event service mechanisms for terrestrial- and satellite-based telecommunication switch management) and integrated database/high-speed communication systems (such as wide-area video-on-demand servers). These types of systems also benefit from a flexible framework that automates and simplifies the dynamic configuration and parallel execution of their distributed services.

My current research involves extending the `ASX` framework to use parallel processing, along with various filter composition techniques, to optimize event filtering [Sch94c] for dynamic multi-point (DMP) applications. Examples of DMP applications include satellite telemetry processing systems, fault management in large-scale network management systems, real-time market data analysis systems, on-line news "clipping" services, and distributed agents for mobile personal communication systems. Event filtering is a data reduction mechanism that eliminates unnecessary network traffic and unnecessary processing at consumer endsystems in DMP systems. In addition, filtering is used to demultiplex and classify events, which supports network monitoring and automated fault management.

In the next five years, I envision that my research will involve experimentation-based techniques for integrating distributed computing and end-system parallel processing to address applications like tele-conferencing, scientific visualization, medical imaging, and global personal communication systems in a gigabit network environment. Ideally, I would like to become a participant in a gigabit testbed environment that would enable me to conduct my research in both wide-area and local-area settings.

Bibliography

[ABG+86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activiation: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, pages 53–79, February 1992.

[ACR88] M. Stella Atkins, Samuel T. Chanson, and James B. Robinson. LNTP – An Efficient Transport Protocol for Local Area Networks. In *Proceedings of the Conference on Global Communications (GLOBECOM)*, pages 705–710, 1988.

[Atk88] M. Stella Atkins. Experiments in SR with Different Upcall Program Structures. *ACM Transactions on Computer Systems*, 6(4):365–392, November 1988.

[Bir89] Andrew D. Birrell. An Introduction to Programming with Threads. Technical Report SRC-035, Digital Equipment Corporation, January 1989.

[Bja90] Bjarne Stroustrup and Margret Ellis. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Bja91] Bjarne Stroustrup. *The C++ Programming Language, 2^{nd} Edition*. Addison-Wesley, 1991.

[BL88] Ronald E. Barkley and T. Paul Lee. A Heap-Based Callout Implementation to Meet Real-Time Needs. In *Proceedings of the USENIX Summer Conference*, pages 213–222. USENIX Association, June 1988.

[BO92] Don Batory and Sean W. O'Malley. The Design and Implementation of Hierarchical Software Systems Using Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4), October 1992.

[Boo93] Grady Booch. *Object Oriented Analysis and Design with Applications (2^{nd} Edition)*. Benjamin/Cummings, Redwood City, California, 1993.

[BSS93] Donald F. Box, Douglas C. Schmidt, and Tatsuya Suda. ADAPTIVE: An Object-Oriented Framework for Flexible and Adaptive Communication Protocols. In *Proceedings of the 4^{th} IFIP Conference on High Performance Networking*, pages 367–382, Liege, Belgium, 1993. IFIP.

[BZ93] Torsten Braun and Martina Zitterbart. Parallel Transport System Design. In *Proceedings of the 4^{th} IFIP Conference on High Performance Networking*, Belgium, 1993. IFIP.

[CDJM91] Ramon Caceres, Peter Danzig, Sugih Jamin, and Danny Mitzel. Characteristics of Wide-Area TCP/IP Conversations. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, pages 101–112, Zurich Switzerland, September 1991. ACM.

[Che87] David R. Cheriton. UIO: A Uniform I/O System Interface for Distributed Systems. *ACM Transactions on Computer Systems*, 5(1):12–46, February 1987.

[Cla85] David D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the* 10^{th} *Symposium on Operating System Principles*, Shark Is., WA, 1985.

[CRJ87] Roy Campbell, Vincent Russo, and Gary Johnson. The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–126. USENIX Association, November 1987.

[CS91] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP Vol II: Design, Implementation, and Internals*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[CSSZ90] Eric C. Cooper, Peter A. Steenkiste, Robert D. Sansom, and Brian D. Zill. Protocol Implementation on the Nectar Communication Processor. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, pages 135–144, Philadelphia, PA, September 1990. ACM.

[CT90] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208, Philadelphia, PA, September 1990. ACM.

[Cus93] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 1993.

[CWWS92] Jon Crowcroft, Ian Wakeman, Zheng Wang, and Dejan Sirovica. Is Layering Harmful? *IEEE Network Magazine*, January 1992.

[DAPP93] Peter Druschel, Mark B. Abbott, Michael Pagels, and Larry L. Peterson. Network subsystem design. *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, 7(4), July 1993.

[DDK+90] Willibald Doeringer, Doug Dykeman, Matthias Kaiserswerth, Bernd Meister, Harry Rudin, and Robin Williamson. A Survey of Light-Weight Transport Protocols for High-Speed Networks. *IEEE Transactions on Communication*, 38(11):2025–2039, November 1990.

[EKB+92] J.R. Eykholt, S.R. Kleiman, S. Barton, R. Faulkner, A Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyond Multiprocessing... Multithreading the SunOS Kernel. In *Proceedings of the Summer USENIX Conference*, San Antonio, Texas, June 1992.

[Fel90] David C. Feldmeier. Multiplexing Issues in Communications System Design. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, pages 209–219, Philadelphia, PA, September 1990. ACM.

[FM92] D.C. Feldmeier and A.J. McAuley. Reducing Ordering Constraints to Improve Performance. In *Proceedings of the* 3^{rd} *IFIP Workshop on Protocols for High-Speed Networks*, Stockholm, Sweden, May 1992.

[Gar90] Arun Garg. Parallel STREAMS: a Multi-Process Implementation. In *Proceedings of the Winter USENIX Conference*, Washington, D.C., January 1990.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994.

[GKWW89] D. Giarrizzo, M. Kaiserswerth, T Wicki, and R. Williamson. High-Speed Parallel Protocol Implementations. In *Proceedings of the 1st International Workshop on High-Speed Networks*, pages 165–180, May 1989.

[GNI92] M. Goldberg, Gerald Neufeld, and Mabo Ito. A Parallel Approach to OSI Connection-Oriented Protocols. In *Proceedings of the* 3^{rd} *IFIP Workshop on Protocols for High-Speed Networks*, Stockholm, Sweden, May 1992.

[Haa91] Zygmunt Haas. A Protocol Structure for High-Speed Communication Over Broadband ISDN. *IEEE Network Magazine*, pages 64–70, January 1991.

[HMPT89] Norman C. Hutchinson, Shivakant Mishra, Larry L. Peterson, and Vicraj T. Thomas. Tools for Implementing Network Protocols. *Software Practice and Experience*, 19(9):895–916, September 1989.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The *x*-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[Jac88] Van Jacobson. Congestion Avoidance and Control. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, Stanford, Calif., August 1988.

[JBB92] Van Jacobson, Robert Braden, and David Borman. TCP Extensions for High Performance. *Network Information Center RFC 1323*, pages 1–37, October 1992.

[JF88] R. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(5):22–35, June/July 1988.

[JSB90] Jiraj Jain, Mischa Schwartz, and Theodore Bashkow. Transport Protocol Processing at GBPS Rates. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, pages 188–199, Philadelphia, PA, September 1990. ACM.

[KC88] Hemant Kanakia and David R. Cheriton. The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, pages 175–187, Stanford, CA, August 1988. ACM.

[LC87] Mark A. Linton and Paul R. Calder. The Design and Implementation of InterViews. In *Proceedings of the USENIX C++ Workshop*, November 1987.

[LKAS93] Bert Lindgren, Bobby Krupczak, Mostafa Ammar, and Karsten Schwan. Parallelism and Configurability in High Performance Protocol Architectures. In *Proceedings of the Second Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Williamsburg, Virgina, September 1993. IEEE.

[LMKQ89] S. J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[Mat93] Mats Bjorkman and Per Gunningberg. Locking Strategies in Multiprocessor Implementations of Protocols. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, San Francisco, California, 1993. ACM.

[MB91] J. C. Mogul and A. Borg. The Effects of Context Switches on Cache Performance. In *Proceedings of the* 4^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Santa Clara, CA, April 1991. ACM.

[MD91] Paul E. McKenney and Ken F. Dove. Efficient Demultiplexing of Incoming TCP Packets. Technical Report SQN TR92-01, Sequent Computer Systems, Inc., December 1991.

[MJ93] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter USENIX Conference*, pages 259–270, San Diego, CA, January 1993.

[MK91] Maria D. Maggio and David W. Krumme. A Flexible System Call Interface for Interprocess Communication in a Distributed Memory Multicomputer. *Operating Systems Review*, 25(2):4–21, April 1991.

[MS92] H. E. Meleis and D. N. Serpanos. Memory Management in High-Speed Communication Subsystems. In *Proceedings of the* 1^{st} *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, February 1992.

[NYKT94] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Performance Issues in Parallelized Network Protocols. In *Submission to the Operating Systems Design and Implementation conference*. USENIX Association, November 1994.

[OP92] Sean W. O'Malley and Larry L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[PBS89] Larry L. Peterson, Nick Buchholz, and Richard D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[PR90] D. L. Presotto and D. M. Ritchie. Interprocess Communication in the Ninth Edition UNIX System. *UNIX Research System Papers, Tenth Edition*, 2(8):523–530, 1990.

[Pre93] David Presotto. Multiprocessor Streams for Plan 9. In *Proceedings of the United Kingdom UNIX User Group Summer Proceedings*, London, England, January 1993.

[PS91] Thomas F. La Porta and Mischa Schwartz. Architectures, Features, and Implementation of High-Speed Transport Protocols. *IEEE Network Magazine*, pages 14–22, May 1991.

[PS93] Thomas La Porta and Mischa Schwartz. Performance Analysis of MSP: a Feature-Rich High-Speed Transport Protocol. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, San Francisco, California, 1993. IEEE.

[Rag93] Steve Rago. *UNIX System V Network Programming*. Addison-Wesley, Reading, MA, 1993.

[Rit84] Dennis Ritchie. A Stream Input–Output System. *AT&T Bell Labs Technical Journal*, 63(8):311–324, October 1984.

[SBS93] Douglas C. Schmidt, Donald F. Box, and Tatsuya Suda. ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment. *Journal of Concurrency: Practice and Experience*, 5(4):269–286, June 1993.

[Sch92] Douglas C. Schmidt. IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services. *C++ Report*, 4(9), November/December 1992.

[Sch93a] Douglas 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*, 5(7), September 1993.

[Sch93b] Douglas C. Schmidt. The Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing (Part 1 of 2). *C++ Report*, 5(2), February 1993.

[Sch94a] Douglas C. Schmidt. A Domain Analysis of Network Daemon Design Dimensions. *C++ Report*, 6(3), March/April 1994.

[Sch94b] Douglas C. Schmidt. ASX: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994. USENIX Association.

[Sch94c] Douglas C. Schmidt. High-Performance Event Filtering for Dynamic Multi-point Applications. In 1^{st} *Workshop on High Performance Protocol Architectures (HIPPARCH)*. INRIA, December 1994.

[Sch94d] Douglas C. Schmidt. Transparently Parameterizing Synchronization Mechanisms into a Concurrent Distributed Application. *C++ Report*, 6(6), July/August 1994.

[Sch95] Douglas C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Programs*, Reading, MA, June 1995. Addison-Wesley.

[SP90] J. Sterbenz and G. Parulkar. AXON: Application-Oriented Lightweight Transport Protocol Design. In *International Conference on Computers and Communications*, New Delhi, India, November 1990.

[SPY+93] Sunil Saxena, J. Kent Peacock, Fred Yang, Vijaya Verma, and Mohan Krishnan. Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes. In *Proceedings of the Winter USENIX Conference*, pages 85–106, San Diego, CA, January 1993.

[SS93] Douglas C. Schmidt and Tatsuya Suda. Transport System Architecture Services for High-Performance Communications Systems. *IEEE Journal on Selected Areas in Communication*, 11(4):489–506, May 1993.

[SS94a] Douglas C. Schmidt and Tatsuya Suda. An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems. *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, 2:280–293, December 1994.

[SS94b] Douglas C. Schmidt and Tatsuya Suda. Experiences with an Object-Oriented Architecture for Developing Extensible Distributed System Management Software. In *Proceedings of the Conference on Global Communications (GLOBECOM)*, San Francisco, CA, November/December 1994. IEEE.

[SS94c] Douglas C. Schmidt and Tatsuya Suda. Measuring the Impact of Alternative Parallel Process Architectures on Communication Subsystem Performance. In *Proceedings of the* 4^{th} *International Workshop on Protocols for High-Speed Networks*, Vancouver, British Columbia, August 1994. IFIP.

[SS94d] Douglas C. Schmidt and Tatsuya Suda. The Service Configurator Framework: An Extensible Architecture for Dynamically Configuring Concurrent, Multi-Service Network Daemons. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, pages 190–201, Pittsburgh, PA, March 1994. IEEE.

[SS95a] Douglas C. Schmidt and Tatsuya Suda. Measuring the Performance of Alternative Process Architectures for Parallelizing Communication Subsystems. *Submitted to the IEEE/ACM Journal of Transactions on Networking*, 1995.

[SS95b] Douglas C. Schmidt and Tatsuya Suda. Measuring the performance of parallel message-based process architectures. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, Boston, MA, April 1995. IEEE.

[SSS+93] Douglas C. Schmidt, Burkhard Stiller, Tatsuya Suda, Ahmed Tantawy, and Martina Zitterbart. Language Support for Flexible, Application-Tailored Protocol Configuration. In *Proceedings of the* 18^{th} *Conference on Local Computer Networks*, pages 369–378, Minneapolis, Minnesota, September 1993.

[Ste90] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, Reading, Massachusetts, 1992.

[Sun87] Sun Microsystems. XDR: External Data Representation Standard. *Network Information Center RFC 1014*, June 1987.

[Sun92] Sun Microsystems. *Network Interfaces Programmer's Guide*, Chapter 6 (TLI Interface) edition, 1992.

[Svo89] Liba Svobodova. Implementing OSI Systems. *IEEE Journal on Selected Areas in Communications*, SAC-7:1115–1130, September 1989.

[Ten89] David L. Tennenhouse. Layered Multiplexing Considered Harmful. In *Proceedings of the* 1^{st} *International Workshop on High-Speed Networks*, May 1989.

[TRG+87] Avadis Tevanian, Richard Rashid, David Golub, David Black, Eric Cooper, and Michael Young. Mach Threads and the Unix Kernel: The Battle for Control. In *Proceedings of the USENIX Summer Conference*. USENIX Association, August 1987.

[USN84] USNA. *TTCP: a test of TCP and UDP Performance*, Dec 1984.

[VL87] George Varghese and Tony Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *The Proceedings of the* 11^{th} *Symposium on Operating System Principles*, November 1987.

[VZ91] Raj Vaswani and John Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *Proceedings of the* 13^{th} *Symposium on Operating System Principles*, pages 26–40, Pacific Grove, CA, October 1991. ACM.

[WBC+93] G. Watson, D. Banks, C. Calamvokis, C. Dalton, A. Edwards, and J. Lumley. Afterburner. *IEEE Network Magazine*, 7(4), July 1993.

[WF93] C. Murray Woodside and R. Greg Franks. Alternative Software Architectures for Parallel Protocol Execution with Synchronous IPC. *IEEE/ACM Transactions on Networking*, 1(2), April 1993.

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++ - an object-oriented application framework in C++. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, pages 46–57. ACM, September 1988.

[WM87] Richard W. Watson and Sandy A. Mamrak. Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices. *ACM Transactions on Computer Systems*, 5(2):97–120, May 1987.

[WM89] C. Murray Woodside and J. Ramiro Montealegre. The Effect of Buffering Strategies on Protocol Execution Performance. *IEEE Transactions on Communications*, 37(6):545–554, June 1989.

[Zit91] Martina Zitterbart. High-Speed Transport Components. *IEEE Network Magazine*, pages 54–63, January 1991.

[ZJ91] Jonathan M. Zweig and Ralph Johnson. Delegation in C++. *Journal of Object-Oriented Programming*, 4(7):31–34, November/December 1991.

[ZS90] Xi Zhang and Aruna Seneviratne. An Efficient Implementation of High-Speed Protocol without Data Copying. In *Proceedings of the* 15^{th} *Conference on Local Computer Networks*, pages 443–450, Minneapolis, MN, October 1990. IEEE.

[ZST93] Martina Zitterbart, Burkhard Stiller, and Ahmed Tantawy. A Model for High-Performance Communication Subsystems. *IEEE Journal on Selected Areas in Communication*, 11(4):507–519, May 1993.

[Zwe90] Jonathan M. Zweig. The Conduit: a Communication Abstraction in C++. In *Proceedings of the* 2^{nd} *USENIX C++ Conference*, pages 191–203. USENIX Association, April 1990.