ADAPTIVE
A Framework for Experimenting with High-Performance Transport System Process Architectures

Douglas C. Schmidt and Tatsuya Suda
schmidt@ics.uci.edu and suda@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine, California 92717

An earlier version of this paper appeared in the proceedings of the second International Conference on Computer Communication Networks in San Diego, California, June 1993.

Abstract

Recent advances in VLSI and fiber optic technology are shifting application performance bottlenecks from the underlying networks to the transport system and higher-layer communication protocols. Developing process architectures that effectively utilize multi-processing is one promising technique for alleviating these performance bottlenecks. This paper describes a flexible framework called ADAPTIVE that supports the development of, and experimentation with, process architectures for multi-processor platforms. ADAPTIVE provides a modular, object-oriented framework that generates application-tailored protocol configurations and maps these configurations onto suitable process architectures that satisfy multimedia application performance requirements on high-speed networks. This paper describes several alternative process architectures and outlines the techniques used in ADAPTIVE to support controlled experimentation with these alternatives.

1 Introduction

Transport systems must undergo significant changes to meet the performance requirements of the increasingly demanding and diverse multimedia applications that will run on the next generation of high-speed networks. Transport systems combine protocol processing tasks (such as connection management, data transmission control, remote context management, error protection, and presentation conversions) together with operating system services (such as memory and process management) and hardware devices (such as high-speed network controllers) to support diverse applications running on diverse local, metropolitan, and wide area networks.

Application performance is significantly affected by the *process architecture* of the transport system [1, 2, 3]. A process architecture binds certain communication protocol entities (such as layers, tasks, connections, and/or messages) together with logical and/or physical processing elements. This paper describes a flexible framework called ADAPTIVE that supports, among other things, development and controlled experimentation with alternative process architectures. A major objective of the ADAPTIVE project is to determine process architectures that effectively utilize parallelism to satisfy multimedia application performance requirements on high-speed networks.

The paper is organized as follows: Section 2 motivates the need for research on high-performance transport systems; Section 3 briefly summarizes the architectural design of the ADAPTIVE transport system; Section 4 outlines several alternative process architectures; Section 5 discusses ADAPTIVE's process architecture support in detail; and Section 6 presents concluding remarks.

2 Research Background

The throughput, latency, and reliability requirements of multimedia applications such as interactive voice, video conferencing, supercomputer visualization, and collaborative work are more stringent and diverse than those found in traditional applications such as remote login or file transfer. However, conventional transport systems possess performance limitations that impede their ability to support multimedia applications running on high-speed networks such as DQDB, FDDI, and ATM-based B-ISDN. Application performance is influenced by a number of transport system factors including (1) process management (such as context switching, synchronization, and scheduling overhead), (2) message management (such as memory-to-memory copying and dynamic buffer allocation), (3) multiplexing and demultiplexing, (4) protocol processing tasks (such as checksumming, segmentation, reassembly, retransmission timer, flow control, connection management, and routing), and (5) network interface hardware [4, 5].

A number of empirical studies have demonstrated that process management and message management are responsible for a significant percentage of the total transport system performance overhead [1, 3, 5, 6, 7, 8]. In general, these sources of transport system overhead have become a

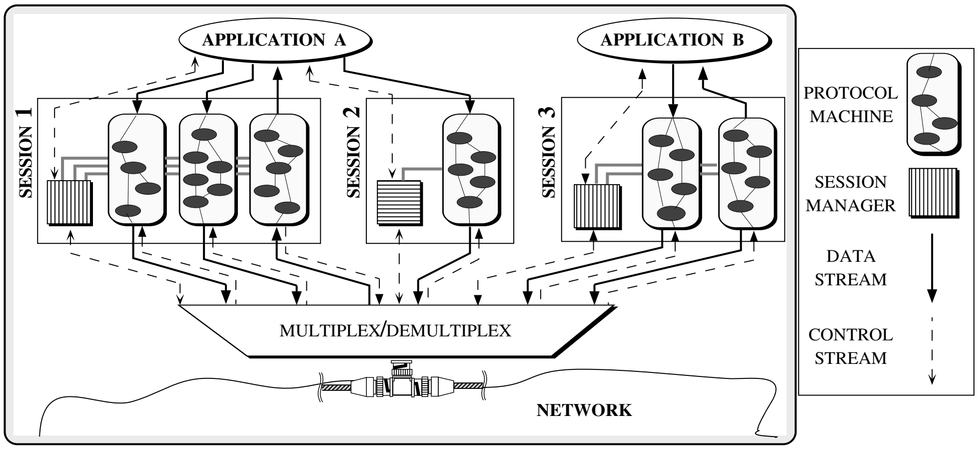

Figure 1: The ADAPTIVE Transport System Architecture and Services

throughput preservation problem as VLSI and fiber optic technologies continually increase network channel speeds. In particular, the bandwidth available from high-speed networks is often reduced by an order of magnitude by the time it is actually delivered to applications [9]. Furthermore, this problem persists despite an increase in computer CPU speeds and memory bandwidth [10]. For example, network channel speeds have increased by 5 or 6 orders of magnitude (from kbps to Gbps), whereas CPU speeds and memory bandwidth have increased by 2 or 3 orders of magnitude (from 1 MIP up to 100 MIPS for CPUs and 100ns down to 10ns access times for high-speed cache memory) [11].

Developing process architectures that effectively utilize multi-processing is a promising technique for alleviating the throughput preservation problem. However, designing and implementing transport systems that utilize parallelism efficiently is a complex, challenging task. Therefore, this paper describes a flexible framework called ADAPTIVE that simplifies the development of process architectures that utilize multi-processing. These process architectures include (1) *Layer Parallelism*, which associates a "process-per-protocol-layer" (such as presentation layer, transport layer, and network layer), (2) *Task Parallelism*, which associates a "process-per-protocol-task" (such as flow control, segmentation and reassembly, error detection, and routing), (3) *Connectional Parallelism*, which associates a "process-per-connection," and (4) *Message Parallelism*, which associates a "process-per-message."

The ADAPTIVE framework also facilitates experimentation with the various process architecture alternatives. Several studies have compared the advantages and disadvantages of these process architectures via qualitative analysis [2, 12]. However, few studies have quantitatively compared the performance of the alternative process architectures via controlled, empirical experimentation. In particular, existing research that measures the performance of process architectures focuses on only one or two approaches [7, 9, 13, 14]. Moreover, these empirical studies typically do not control for critical confounding factors such as hardware platform, operating system, and protocol implementation. By not controlling for these factors, it is difficult to isolate and accurately assess the performance impacts of a particular process architecture. The ADAPTIVE system, on the other hand, is designed to provide a controlled environment for experimenting with alternative process architectures. This enables more precise measurement of a process architecture's impact on various aspects of application and transport system performance.

3 Overview of ADAPTIVE

The ADAPTIVE system is "*A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment*." ADAPTIVE provides an integrated environment for developing and experimenting with flexible transport system architecture services. These services support application-tailored communication protocols for diverse multimedia applications running on high-performance networks. To enhance service flexibility, ADAPTIVE maintains a collection of reusable "building block" protocol mechanisms that may be automatically composed together and instantiated based upon specifications of application requirements. To enhance performance, the generated protocols may execute in parallel on several target platforms such as shared memory and message-passing multi-processors. This paper focuses primarily on ADAPTIVE's process architecture support; other aspects of ADAPTIVE are described in [15].

Figure 1 depicts the main levels of abstraction and services in ADAPTIVE's architecture. Multimedia applications that generate and receive various types of synchronized and independent traffic (such as voice, video, text, and image) access ADAPTIVE's services via an interface between the transport system and the end-user applications. This application interface manages local host resources such as I/O descriptors and communication ports. It also provides a queueing point for exchanging application data and control messages with the lower-level transport system components.

The ADAPTIVE transport system maintains a collection of *protocol machines* for each application. A protocol machine is an executable instantiation of a communication protocol that implements a uni-directional *data stream* containing customized *session architecture* service mechanisms such as connection management, error protection, end-to-end flow control, remote context management, presentation services, and routing. Session architecture services perform "end-to-end" and "link-to-link" protocol processing tasks on incoming and outgoing PDUs.

To support layered protocol families such as OSI and TCP/IP, ADAPTIVE aggregates session architecture services into distinct layers of functionality (such as the session, transport, and network layers) via a standard, reusable set of *protocol family architecture* services. These services manage the "layer-to-layer" tasks (such as message management, multiplexing and demultiplexing, and layer-to-layer flow control) that exchange PDUs between the protocol layers (and transport system boundaries) on a local host. In addition, protocol family architecture services also supports de-layered communication models (such as those described in [10, 16, 17]). In this case, the protocol family architecture services operate between the application interface, de-layered transport system, and network interface.

Applications, session architecture services, and protocol family architecture services all execute within a process environment provided by services in the *kernel architecture*. These services manage the process architecture, virtual and physical memory, event timers, and device drivers to provide a portable "software veneer" for hardware devices such as processing elements, primary and secondary storage, hardware clocks, and network controllers (which implement the link-level protocols for various networks such as FDDI, Token Ring, ATM, Ethernet, and DQDB).

To permit meaningful experiments on alternative process architectures, ADAPTIVE is designed to control many of the confounding factors in the transport system. To facilitate this, ADAPTIVE utilizes a modular architecture that decouples the policies and mechanisms in each level of the transport system. This decoupling enables experimenters to hold the higher-level protocol family architecture and session architecture components constant, while varying certain process architecture components and accurately measuring the resulting performance impacts. ADAPTIVE's modularity also increases its portability, allowing it to run in multiple underlying kernel architectures (such as UNIX, Mach, and transputer platforms) and protocol family architectures (such as STREAMS and *x*-kernel). This paper focuses primarily on a version of ADAPTIVE that is hosted the UNIX STREAMS environment [18]. An alternative approach that describes hosting ADAPTIVE in the *x*-kernel is presented in [19].

4 Process Architecture Models

To address the throughput preservation problem, ADAPTIVE provides a flexible framework for developing and experimenting with alternative process architectures. A process architecture binds communication protocol entities to logical and/or physical processing elements (PE). Protocol entities include abstractions such as layers, tasks, connections, and/or messages. Likewise, operating system processes[1] are abstractions of hardware PEs. On a multiprocessor separate processes may execute on multiple PEs, whereas on a uni-processor each process may be "time-sliced" on a single PE.

Regardless of whether multiple or single PEs are used, the process architecture significantly impacts the performance of applications and transport systems. In particular, certain process architectures are capable of exploiting available OS and hardware parallelism more effectively compared with other architectures. For example, certain process architectures increase the overhead of interprocess communication and memory-to-memory copying, whereas others increase the overhead of synchronization and/or context switching. In general, the suitability of a particular process architecture depends on a variety of factors such as (1) the type of traffic generated by applications (such as bursty vs. continuous and short-duration vs. long-duration), (2) the architecture of the hardware and operating system (such as message passing vs. shared memory, lightweight processes vs. heavyweight processes, and micro-kernel vs. macro-kernel), and (3) the underlying network environment (such as high-speed vs. low-speed and large frame size vs. small frame size).

This section outlines the distinguishing features of four process architectures supported by ADAPTIVE. These process architectures fall into three general categories: *horizontal*, *vertical*, and *hybrid* [12]. Although each process architecture has different structural and performance characteristics, it is possible to implement the same protocol family functionality (such as the OSI, TCP/IP, and F-CSS [16]) with any approach.

4.1 Horizontal Process Architectures

Horizontal process architectures associate PEs with protocol layers or protocol tasks. Each PE performs certain protocol operations on PDUs that are then exchanged with neighboring PEs. Two common examples of horizontal process architectures are *Layer Parallelism* and *Task Parallelism*.

• **Layer Parallelism:** Layer Parallelism is a coarse-grained horizontal process architecture. As shown in Figure 2 (1), a PE is associated with each protocol layer (such as the session, transport, and network layers) in the protocol stack. Messages flow through the layers in a coarse-grain "pipelined" manner. Inter-layer buffering, flow control, and

[1] The term "process" is used in this paper to refer to a thread of control executing within an address space. Other systems use different terminology (such as lightweight processes [20] or threads [21]) to denote essentially the same concept.

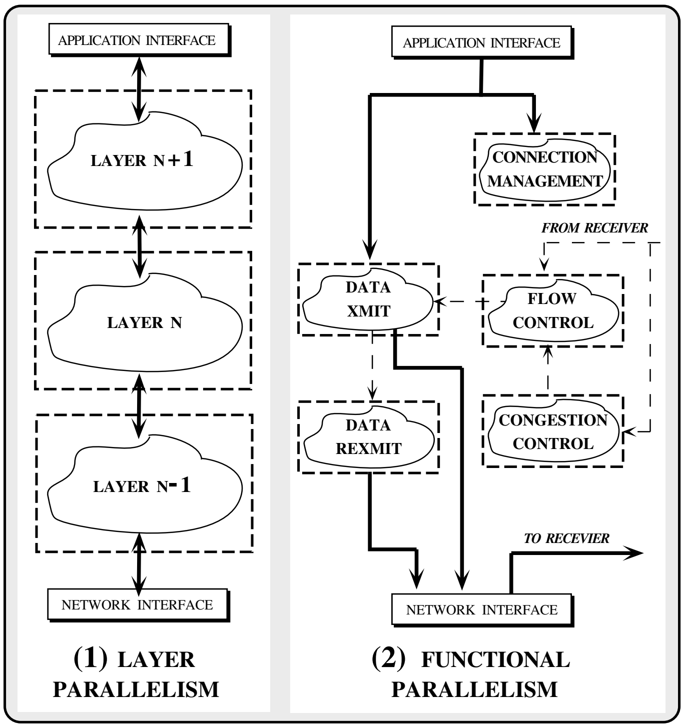

Figure 2: Horizontal Process Architectures

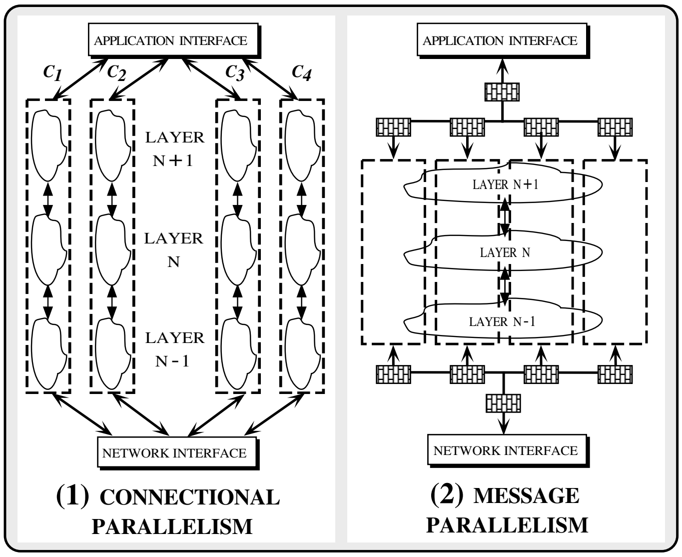

Figure 3: Vertical Process Architectures

"stage balancing" [22] are typically necessary since the processing activities at each layer may not execute at the same rate. The primary advantage of Layer Parallelism is the simplicity of its design, which corresponds closely to standard layered communication architecture specifications [23]. In addition, it is also suitable for systems that possess a limited number of PEs. The primary disadvantages of this approach are (1) its fixed amount of parallelism, which is limited by the number of protocol layers, (2) the high synchronization and communication overhead required to move messages between layers, and (3) limited support for PE load balancing since PEs are dedicated to specific protocol layers.

• **Task Parallelism:** Task Parallelism is a fine-grained horizontal process architecture. This approach utilizes multiple PEs to perform many protocol processing tasks in parallel via a "pipeline" [9]. Common protocol tasks include (1) connection management (*e.g.,* connection establishment and termination), (2) header composition and decomposition (*e.g.,* address resolution and demultiplexing), (3) PDU-level and bit-level error protection (*e.g.,* detecting, reporting, and retransmitting out-of-sequence PDUs and computing checksums), (4) segmentation and reassembly, (5) routing, and (6) flow control. Figure 2 (2) illustrates a fine-grain pipeline configuration where multiple PEs execute individual protocol tasks on messages flowing through the sender-side and receiver-side of a protocol session. The primary advantages of this approach are (1) the potential performance improvements from using multiple PEs and (2) the ability to substitute alternative mechanisms for certain protocol tasks [16]. However, the disadvantages are that (1) careful programming is required to minimize the memory contention and synchronization overhead resulting from the communication between separate PEs, (2) load balancing is difficult, and (3) non-standard, "de-layered" communication models are typically required to increase the number of tasks available for parallel execution [10, 16].

4.2 Vertical Process Architectures

Vertical process architectures associate OS processes with connections and messages rather than with protocol layers or tasks [2, 7]. This approach assigns a separate process to escort incoming and outgoing messages through the protocol stack, delivering messages "down" to network interfaces or "up" to applications. Two examples of vertical process architectures are *Connectional Parallelism* and *Message Parallelism*.

• **Connectional Parallelism:** Connectional Parallelism is a coarse-grain vertical process architecture that dedicates a separate PE for each connection. Figure 3 (1) illustrates this approach, where connections C_1, C_2, C_3, and C_4 are bound to separate PEs that process all messages associated with their connection. This approach is useful for network servers that handle many open connections simultaneously. The advantages of Connectional Parallelism are (1) inter-layer communication overhead is reduced (since moving between protocol layers may not require a context switch), (2) synchronization and communication overhead is relatively low within a given connection (since synchronous *intra-process* subroutine calls and upcalls [24] may be used to communicate between the protocol layers), and (3) the amount of available parallelism is determined dynamically (rather than statically) since it is a function of the number of active connections rather than the number of layers or tasks. One disadvantage with Connectional Parallelism is the difficulty of PE load balancing. For example, a highly active connection may swamp its PE with messages, leaving other PEs tied up at less active or idle connections. In addition, to increase the opportunity for exploiting parallelism, packet filters[2] are typically required for Connectional Parallelism since the network interface must demultiplex on the basis of PDU address information (such as connection identifiers, port numbers, or IP addresses) that is actually associated with protocols residing several layers "above" in a protocol stack.

• **Message Parallelism:** Message Parallelism is a fine-grain vertical process architecture that associates a separate PE with every incoming or outgoing message. Each message is typically stored in a buffer residing in shared memory. As illustrated in Figure 3 (2), a pointer to the message is passed to the next available PE, which performs all the protocol processing tasks on that message. The advantages of Message Parallelism are similar to those for Connectional Parallelism. Moreover, the degree of available parallelism may be higher since it depends on the number of messages exchanged, rather than the number of connections. Likewise, processing loads may be balanced more evenly between PEs since each incoming message may be dispatched to an available PE. The primary disadvantages of Message Parallelism are the overhead resulting from (1) resource management and scheduling support necessary to associate a process-per-message and (2) synchronization and mutual exclusion primitives required to serialize access to shared resources (such as memory buffers and control blocks that reassemble protocol segments addressed to the same higher-layer session).

In general, the coarse-grained Layer Parallelism and Connectional Parallelism process architectures are simpler to design and implement than the fine-grain Message Parallelism and Task Parallelism architectures since there is less interaction between the PEs. However, the coarse-grain approaches typically provide less parallelism, which limits their scalability. For example, Layer Parallelism possesses a small, fixed amount of parallelism determined by the number of protocol layers. The parallelism available in the Message Parallelism approach, on the other hand, is a function of the number of messages, which may be much larger than the number of layers, tasks, and/or connections. Note that there are hybrid approaches that combine horizontal and vertical process architectures. For instance, it is possible to assign multiple PEs to each connection, thereby combining Task Parallelism and Connectional Parallelism [22]. This composite approach requires a large number of PEs, special contention-free memory, and careful programming to significantly improve performance, however.

[2] Packet filters [25] are devices that allow applications and higher-level protocols to "program" the 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.

5 Process Architecture Support in ADAPTIVE

This section describes the various components available in ADAPTIVE for developing and experimenting with alternative process architectures. Section 5.1 briefly describes ADAPTIVE's resource and tool components and outlines how they interact with its process architecture framework. Section 5.2 describes ADAPTIVE's process architecture framework in detail.

5.1 Resource and Tool Components in ADAPTIVE

ADAPTIVE contains an integrated set of resource and tool components that automate many steps required to generate and execute application-tailored protocol machines (illustrated in Figure 4). In the generation stage, ADAPTIVE's tools may be used to create executable protocol machines that are customized for the performance requirements of applications (or classes of applications). In the execution stage, applications invoke the previously generated protocol machines to perform their data transport activities efficiently. In addition, if the preconfigured collection of protocol machines is inadequate, applications may adaptively "fine-tune" protocol machine functionality at run-time using ADAPTIVE's reconfiguration services [26]. The primary resources and tools used in the generation and execution stages are described below.

5.1.1 Protocol Machine Generation Stage

As shown in Figure 4 (1), the generation stage transforms descriptions of protocol machine functionality (called *protocol machine configurations*) into executable *protocol machine instantiations*. A protocol machine configuration describes the peer-to-peer tasks (such as connection management, segmentation, or duplicate control) and ordered interrelations between tasks (such as "perform resequencing before reassembly" or "compute checksum before flow control") that process PDUs flowing through the protocol machine. A protocol machine configuration is submitted to the *synthesizer tools*. These tools perform syntactic and semantic analysis, optimization, and code generation to produce a protocol machine instantiation that may be specially-tailored for a particular process architecture. Each protocol machine implements the minimal set of functionality required to process a uni-directional stream of PDUs. Protocol machines are composed of C++ objects residing in a *protocol resource pool*. This resource pool contains reusable implementations of various mechanisms for connection establishment, retransmission, data transmission control, remote context management, demultiplexing, event timing, message management, routing, and presentation services. The protocol processing mechanisms in each protocol machine may be customized for (1) the quantitative and qualitative require-

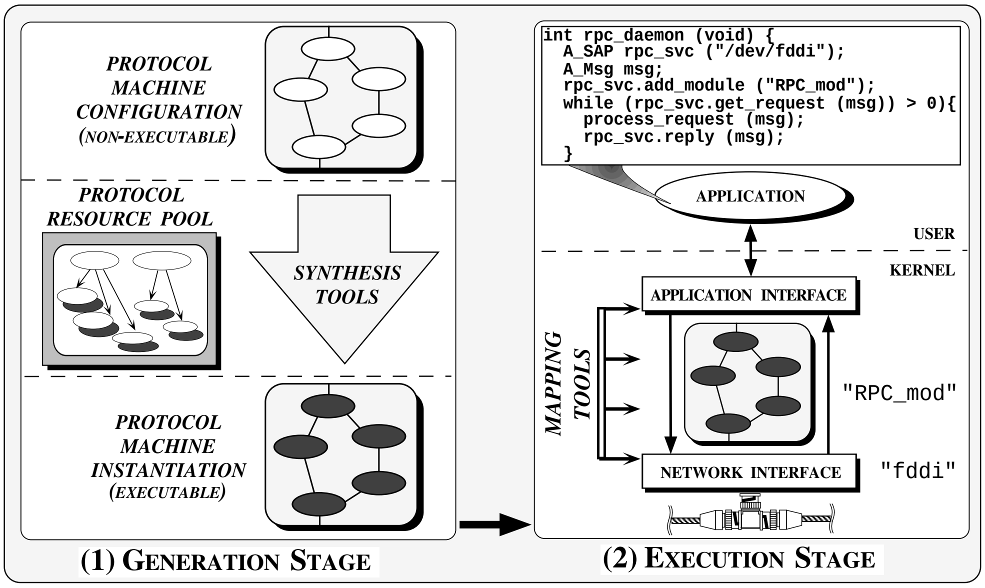

Figure 4: ADAPTIVE Protocol Machine Generation and Execution Stages

ments of the application and (2) the underlying network capabilities [27].

5.1.2 Protocol Machine Execution Phase

Figure 4 (2) illustrates several activities performed by applications and ADAPTIVE transport system components at run-time. As shown in the upper half of the figure, applications open certain communication devices (such as an FDDI controller) and dynamically insert and/or configure protocol machines (such as a machine that implements a Remote Procedure Call (RPC) protocol) via ADAPTIVE's application service interface. This service interface supports the insertion, removal, and/or modification of services at run-time via user-level or kernel-level commands.

As shown in the lower half of Figure 4, various *mapping tools* are responsible for loading user-specified protocol machine(s) into the ADAPTIVE run-time environment. When a protocol machine is selected and invoked by an application, the mapping tools perform a sequence of operations that (1) allocate and initialize the appropriate system control blocks, (2) dynamically link protocol machine mechanisms into the kernel address space and interconnect them to form complete protocol machines, and (3) map selected portions of the protocol machine onto one or more operating system processes. Depending on the underlying operating system and hardware platform, these processes may be mapped onto one or more hardware processing elements.

During the data transfer phase, protocol machines process PDUs that are sent by applications and/or received from network interfaces. If applications or the transport system reconfigure the functionality of a protocol machine at run-time, the mapping tools are invoked to perform the necessary modifications. This enables protocol machines to adapt dynamically to changes in application requirements, transport system resources, and network characteristics [26].

5.2 Process Architecture Components

This section focuses on techniques for implementing ADAPTIVE's process architecture framework within the protocol family architecture and kernel architecture provided by STREAMS [18]. To avoid extraneous development effort, ADAPTIVE is initially being hosted in several existing operating environments including STREAMS in UNIX. The generation and execution components described in Section 5.1 utilize various STREAMS features and capabilities. However, these components are designed in a modular manner to minimize their dependency on the underlying platform. This modularity facilitates (1) controlled experimentation with different configurations of protocol machines and process architectures and (2) provides additional portability and platform transparency for protocol developers. The remainder of this section briefly summarizes the primary components in STREAMS and explains how the ADAPTIVE components described in Section 5.1 are implemented by the STREAMS components.

5.2.1 Overview of STREAMS

STREAMS provides ADAPTIVE with modular protocol family and kernel architectures that possess components with uniform interfaces [18]. As shown in Figure 5, STREAMS components include STREAM *heads*, STREAM *modules*, STREAM *multiplexors*, and STREAM *drivers*. STREAM heads provide a queueing point that segments and reassembles application data into discrete messages. These messages are passed "downstream" from a STREAM head through zero or more modules and/or multiplexers to a STREAM

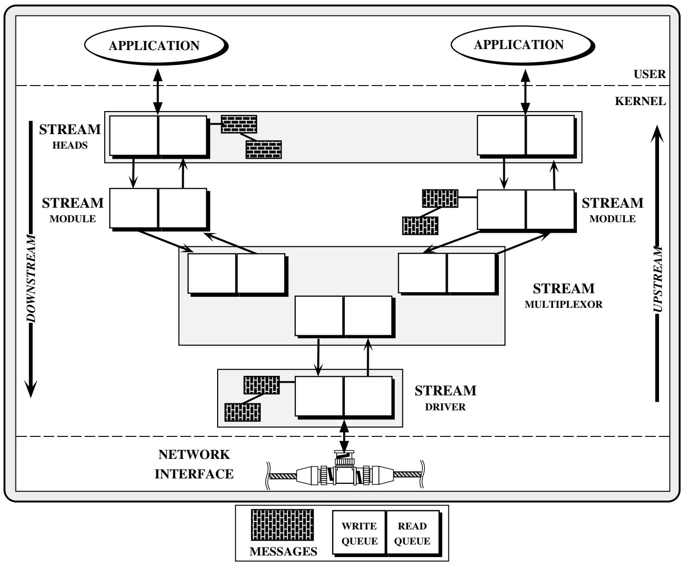

Figure 5: System V STREAMS Architecture

driver that transmits them to the appropriate underlying network. Likewise, drivers receive messages from the network. These messages are passed "upstream" through modules and/or multiplexors to a STREAM head, which coalesces the messages into buffers provided by applications.

STREAM modules and multiplexors implement protocol processing services such as encryption, compression, reliable message delivery, and routing. A STREAM module is linked together with its two adjacent components by a single pair of *read* and *write* queues. A STREAM multiplexor, on the other hand, may be linked together with one or more adjacent components via multiple pairs of read and write queues. Read queues process messages arriving from network devices and write queues process messages generated by applications. Queues may contain a linked list of control and data messages that are stored in "priority-order" and passed between modules and/or multiplexors. The overhead of passing messages between modules and multiplexors is minimized by passing pointers to messages rather than copying the data directly.

Queues contain several standard subroutines that implement either *immediate* and/or *deferred* message processing. Immediate processing is performed by the `put()` subroutine. This subroutine is run when synchronous or asynchronous events occur at a particular queue (such as when an application sends a message downstream or a message arrives on a network interface). Protocol processing operations that must be invoked immediately (such as handling high-priority TCP "urgent data" messages) are performed by `put()`. Deferred processing is performed by the `service()` subroutine. This subroutine is used for protocol operations that do not execute in short, fixed amounts 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).

To enhance performance, the STREAMS components described above execute within the operating system kernel. To enhance flexibility, these components may be linked together dynamically from user-level or kernel-level to form protocol suites such as those specified by the Internet or ISO OSI reference models. For example, modules and multiplexors may be inserted and/or removed dynamically between a STREAM head and a STREAM driver at run-time.

5.2.2 Implementing Alternative Process Architectures in STREAMS

By associating OS processes with different configurations of modules and multiplexors, several process architectures may be implemented in STREAMS. Recent implementations of STREAMS [20] utilize shared memory multi-processors and support various types of parallelism, ranging from fine-grain parallelism controlled by explicit use of kernel synchronization primitives to various synchronization models that are supported automatically by the STREAMS framework. The STREAMS-based version of ADAPTIVE supports Layer Parallelism, Task Parallelism, and Connectional Parallelism, as well as a hybrid process architecture that combines the Task and Connectional approaches.[3]

A particular process architecture may be selected explicitly by developers and/or implicitly by ADAPTIVE's higher-level tool components. During the generation stage for instance, developers may instrument protocol machines with various synchronization primitives such as mutual exclusion locks, condition variables, counting semaphores, and readers/writer locks. These synchronization constructs enable protocol machine instantiations to execute efficiently and correctly in one or more process architectures with a minimal amount of redevelopment effort. This instrumentation process is facilitated by several C++ features used in the protocol resource pool such as (1) abstract base classes, inheritance, and dynamic binding, (2) parameterized types, (3) transparently extensible free store management, (4) conditional compilation, and (5) member function inlining. For example, protocol machines that run in several cooperating processes may contain C++ protocol mechanism objects that are allocated in shared memory. The use of shared memory enables multiple processes to inspect and/or modify certain protocol mechanisms efficiently. Depending on the process architecture, these protocol mechanism objects may be conditionally compiled to activate the mutual exclusion code required to synchronize multiple accesses and/or updates to these objects.

The following paragraphs describe how ADAPTIVE supports Connectional Parallelism and Task Parallelism via certain STREAMS components. Both examples illustrate a STREAMS-based implementation of the data transfer and reception portions of a reliable connection-oriented protocol. To improve flexibility and adaptivity, the tasks shown in

[3] Supporting Message Parallelism efficiently in STREAMS is difficult since it requires a large number of kernel processes.

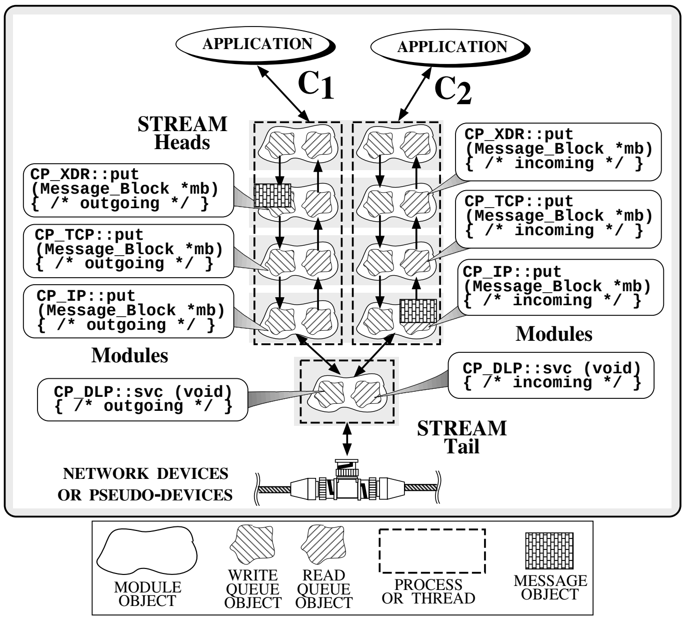

Figure 6: Connectional Parallelism in STREAMS

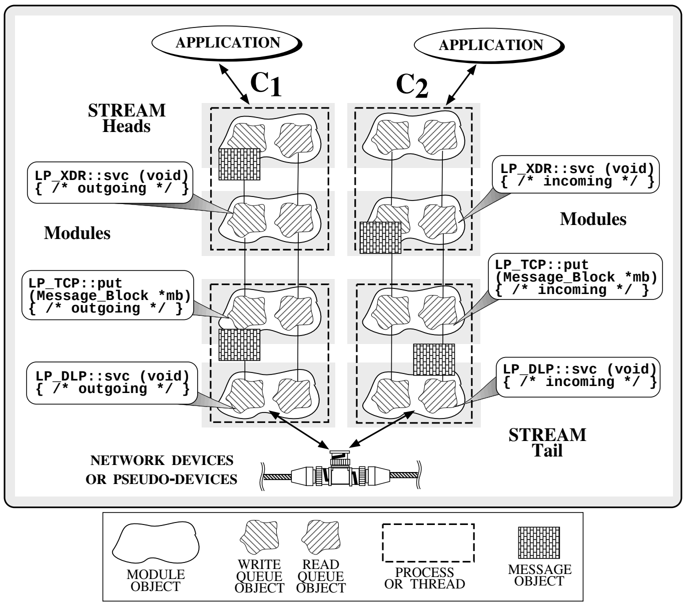

Figure 7: Task Parallelism in STREAMS

the example code are implemented as pointers-to-functions. During protocol generation, the synthesis tools extract the appropriate functions from the protocol resource pool. In addition, the mechanisms that implement these functions may be updated at run-time using a combination of dynamic linking and dynamic binding. This flexibility enables adaptive reconfiguration of protocol machine functionality to account for changes in the run-time environment of the application, transport system, or network.

• **Connectional Parallelism Example:** The protocol machines illustrated in Figure 6 use Connectional Parallelism to associate a "process-per-connection." The write queue subroutines perform all the "outgoing" protocol processing operations on PDUs sent from an application before passing the PDU to the network interface. Likewise, the read queue subroutines perform all the "incoming" protocol processing operations on PDUs received from the network before passing them up to applications.

ADAPTIVE's modularity enables controlled, precise measurement of the performance impact that results from reconfiguring the process architecture. For example, by associating separate processes with the sender-side and receiver-side of each connection (rather than one per-connection, as portrayed in Figure 6), it is possible to extend the process architecture described above to utilize additional parallelism. Note that most of the other session architecture and protocol family architecture factors remain unchanged, however.

• **Task Parallelism Example:** The protocol machines illustrated in Figure 7 use fine-grain Task Parallelism to associate separate processes with certain clusters of protocol tasks. In this approach, processes cooperate in a "pipeline fashion" to operate on multiple incoming and outgoing PDUs in parallel. Each queues' subroutines perform a small number of protocol tasks on each PDU before passing the PDUs to an adjacent queue that may be accessed via a different process.

Depending on the ratio of protocol tasks (which are relatively fixed) to connections (which vary dynamically), this process architecture may utilize a larger amount of available parallelism, compared with the Connection Parallelism approach described above. However, careful programming and "stage balancing" may be required to efficiently coordinate and minimize the overhead of managing the multiple communicating processes [22]. As with the Connectional Parallelism example, it is possible to modify these protocol machines to utilize a more coarse-grain Layer Parallelism process architecture. For instance, rather than associating processes with a small number of application-tailored tasks, certain tasks may be clustered into the standard OSI or TCP/IP protocol layers, with OS processes then associated with the larger clusters.

6 Concluding Remarks

ADAPTIVE is a flexible transport system development environment that addresses the performance requirements of multimedia applications running on high-speed networks. ADAPTIVE provides a framework for experimenting with alternative process architectures in order to help improve protocol performance and reduce transport system overhead. We are currently designing and implementing a prototype implementation of ADAPTIVE written in C++. To experiment with alternative process architectures, this prototype is hosted in the STREAMS framework on a multi-processor

UNIX platform [20]. The multi-processing framework of STREAMS in UNIX provides the basis for developing a number of different process architectures and determining the impact on application and transport system performance in a controlled manner. We are using ADAPTIVE to implement and evaluate a number of protocol machines that are customized for several classes of multimedia applications such as real-time audio and video applications running on several different networks (such as Ethernet, DQDB, and FDDI).

References

[1] D. D. Clark, "Modularity and Efficiency in Protocol Implementation," *Network Information Center RFC 817*, pp. 1–26, July 1982.

[2] J. Jain, M. Schwartz, and T. Bashkow, "Transport Protocol Processing at GBPS Rates," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 188–199, ACM, Sept. 1990.

[3] G. Chesson, "XTP/PE Design Considerations," in *Proceedings of the 1st International Workshop on High-Speed Networks*, May 1989.

[4] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, vol. 27, pp. 23–29, June 1989.

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

[6] R. W. Watson and S. A. Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices," *ACM Transactions on Computer Systems*, vol. 5, pp. 97–120, May 1987.

[7] N. C. Hutchinson and L. L. Peterson, "The *x*-kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, January 1991.

[8] T. F. L. Porta and M. Schwartz, "Architectures, Features, and Implementation of High-Speed Transport Protocols," *IEEE Network Magazine*, pp. 14–22, May 1991.

[9] M. Zitterbart, "High-Speed Transport Components," *IEEE Network Magazine*, pp. 54–63, January 1991.

[10] Z. Haas, "A Protocol Structure for High-Speed Communication Over Broadband ISDN," *IEEE Network Magazine*, pp. 64–70, January 1991.

[11] P. Druschel, M. B. Abbott, M. Pagels, and L. L. Peterson, "Network subsystem design," *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, vol. 7, July 1993.

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

[13] T. L. Porta and M. Schwartz, "Performance Analysis of MSP: a Feature-Rich High-Speed Transport Protocol," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (San Francisco, California), IEEE, 1993.

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

[15] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment," *Journal of Concurrency: Practice and Experience*, vol. 5, pp. 269–286, June 1993.

[16] M. Zitterbart, B. Stiller, and A. Tantawy, "A Model for High-Performance Communication Subsystems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 507–519, May 1993.

[17] D. C. Schmidt, B. Stiller, T. Suda, A. Tantawy, and M. Zitterbart, "Language Support for Flexible, Application-Tailored Protocol Configuration," in *Proceedings of the* 18^{th} *Conference on Local Computer Networks*, (Minneapolis, Minnesota), pp. 369–378, IEEE, Sept. 1993.

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

[19] D. C. Schmidt, "Hosting the ADAPTIVE System in the *x*-Kernel and System V STREAMS," in *Proceedings of the* x-*kernel Workshop*, (Tucson, Arizona), November 1992.

[20] J. Eykholt, S. 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.

[21] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young, "Mach Threads and the Unix Kernel: The Battle for Control," in *Proceedings of the USENIX Summer Conference*, USENIX Association, August 1987.

[22] O. Koufopavlou, A. N. Tantawy, and M. Zitterbart, "Analysis of TCP/IP for High Performance Parallel Implementations," in *17th Conference on Local Computer Networks*, (Minneapolis, Minnesota), Sept. 1992.

[23] M. S. Atkins, "Experiments in SR with Different Upcall Program Structures," *ACM Transactions on Computer Systems*, vol. 6, pp. 365–392, November 1988.

[24] D. D. Clark, "The Structuring of Systems Using Upcalls," in *Proceedings of the* 10^{th} *Symposium on Operating System Principles*, (Shark Is., WA), 1985.

[25] J. C. Mogul, R. F. Rashid, and M. J. Accetta, "The Packet Filter: an Efficient Mechanism for User-level Network Code," in *Proceedings of the* 11^{th} *Symposium on Operating System Principles (SOSP)*, November 1987.

[26] H. K. Huang, T. Suda, G. Takeuchi, and Y. Ogawa, "Protocol Reconfiguration: a Study of Error Handling Mechanisms," in *Proceedings of the* 2^{nd} *International Conference on Computer Communication Networks*, (San Diego, California), ISCA, June 1993.

[27] D. F. Box, D. C. Schmidt, and T. Suda, "ADAPTIVE: An Object-Oriented Framework for Flexible and Adaptive Communication Protocols," in *Proceedings of the* 4^{th} *IFIP Conference on High Performance Networking*, (Liege, Belgium), pp. 367–382, IFIP, 1993.