A Framework for Developing and Experimenting with Parallel Process Architectures to Support High-Performance Transport Systems

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[1]

An earlier version of this paper appeared in the proceedings of the Second High Performance Communication Subsystems Workshop in Williamsburg, Virginia, September 1993.

Abstract

Multi-processing is a promising technique for improving the performance, scalability, and cost effectiveness of communication subsystems. Improving performance is becoming increasingly important to alleviate bottlenecks resulting from network transmission speeds that now often exceed the processing capacity of end-systems. This paper describes a modular framework for developing and experimenting with process architectures for bus-oriented, shared memory multiprocessors. A process architecture binds units of communication protocol processing (such as layers, functions, connections, and messages) with one or more processing elements. This paper describes several alternative process architectures supported by ADAPTIVE and outlines techniques used to perform controlled experimentation with these alternatives.

1 Introduction

Advances in VLSI and fiber optic technology are shifting performance bottlenecks from the underlying networks to the transport system [1]. A transport system consists of *protocol functions* (such as connection management, end-to-end and layer-to-layer flow control, remote context management, segmentation/reassembly, demultiplexing, message buffering, error protection, and presentation conversions), *operating system services* (such as message buffering, asynchronous event invocation, and process management), and *hardware devices* (such as high-speed network adapters) that support distributed applications. Developing high-performance transport systems is essential to support the increasingly demanding throughput and delay requirements of bandwidth-intensive and constrained-latency multimedia applications (such as interactive video conferencing, medical imaging, and scientific visualization [2]). In general, multi-processing has the potential to increase protocol processing rates and reduce delay, for individual sessions, as well as for aggregate end-system performance [3].

Despite a significant increase in the availability of operating system (OS) and hardware platforms that support multi-processing [4, 5, 6, 7], developing transport systems that effectively utilize multi-processing remains a complex and challenging task. This complexity emanates from several application, network, and OS factors. For instance, devising a universally applicable strategy for implementing communication protocols via multiple processing elements (PEs) is complicated by the increasing diversity of application requirements (*e.g.,* high-bandwidth, constrained latency, multimedia data streams, etc.) and network characteristics (*e.g.,* various sizes of packet frames, different channel speeds, etc.). Likewise, designing and implementing robust, flexible, and efficient concurrent software in tightly constrained OS kernel environments remains difficult, due to primitive debugging tools [8], limited memory resources, and subtle synchronization and timing interactions [9].

A wide range of models have been proposed for applying multi-processors to communication protocols [3, 10, 11, 12, 13, 14]. However, existing research has generally not controlled for relevant confounding factors (such as platform architecture, operating system, protocol implementation, application requirements, and network characteristics) that influence the selection and implementation of suitable multi-processor models. One method for systematically measuring these factors is to devise *process architectures* that organize and simplify the development of multi-processor-based transport systems [15]. Process architectures represent a binding between various units of *communication protocol processing* (such as layers, functions, connections, and messages) and various configurations of *OS processes*[2] (which are abstractions of hardware PEs).

The process architecture is one of several factors [15] that influence transport system performance (other factors

[1]This material is based upon work supported by the National Science Foundation under Grant No. NCR-8907909. This research is also supported in part by grants from the University of California MICRO program, Nippon Steel Information and Communication Systems Inc. (ENICOM), Hitachi Ltd., Hitachi America, and Tokyo Electric Power Company.

[2]The term "process" is used in this paper to refer to a flow of control executing within an address space (which may be shared with other processes). Other systems use different terminology (such as lightweight processes [4] or threads [7]) to denote the same basic concepts.

include protocol design and implementation, along with bus, memory, and network interface characteristics). In general, the policies and mechanisms offered by a process architecture significantly affect key sources of application performance overhead such as memory-to-memory copying and data manipulation, process management, and synchronization [12, 16]. In addition, the choice of process architecture also influences demultiplexing strategies [17] and protocol programming techniques [18].

Selecting an appropriate process architecture is an important design decision in domains other than transport systems. For example, event-driven applications (particularly those that route or process messages on the basis of "connection" information) often perform non-communication-related tasks that benefit from a structured approach to multiprocessing [19]. However, our research focuses upon the impact of process architectures on transport system performance since communication protocol behavior and functionality is well-understood and relatively well-defined. Moreover, a large body of literature exists with which to compare our results.

This paper examines a framework for investigating process architecture policies and mechanisms that effectively utilize multi-processors to support applications running on high-speed networks. This framework is part of the ADAPTIVE system [20], which provides an integrated set of tools and resources that simplify and automate certain transport system development and experimentation steps [19]. The paper is organized as follows: Section 2 outlines several alternative process architecture models supported by ADAPTIVE and classifies related work according to the models presented; Section 3 describes how ADAPTIVE implements these process architectures by building upon existing transport system components such as STREAMS [21] and multiprocessor versions of UNIX [4, 5]; and Section 5 discusses concluding remarks.

2 Alternative Process Architectures

Three of the primary components in a multi-processor-based, event-driven transport system are *processing elements* (PEs), which are the underlying protocol execution agents, *data and control messages*, which are sent and received from multiple applications and network devices, and *protocol processing tasks*, which are the protocol-related functions performed upon messages as they arrive and depart. Two fundamental process architecture categories, *Task-based* and *Message-based*, may be discerned by examining the alternative methods they adopt for structuring and combining the primary transport system components. For example, the various Task-based models discussed below structure multiple PEs according to units of protocol functionality, whereas the Message-based models structure the PEs according to the protocol control and data protocol messages received from applications and network interfaces.

Although it is possible to implement any protocol with any model, the process architectures in each category exhibit different structural and performance characteristics. For instance, process architectures exhibit structural characteristics that difference in terms of (1) the granularity of the communication protocol processing unit (*e.g.,* layer, function, connection, message) that executes in parallel, (2) the degree of PE scalability (*i.e.,* fixed versus dynamically scalable), (3) invocation semantics (*e.g.,* synchronous vs. asynchronous) and (4) the effort required to design and implement protocols and services via a process architecture [22]. Likewise, different application, OS and hardware platform, and network environment configurations may interact with different process architectures to yield significantly different performance results [23]. For example, certain process architectures described below exhibit different levels of data movement and context switching overhead.

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

2.1 Task-based Process Architectures

Task-based process architectures associate processes with protocol layers or protocol functions. Two Task-based process architectures supported in ADAPTIVE are *Layer Parallelism* and *Functional Parallelism*. The primary difference between these two models involve the granularity of the protocol processing tasks (*i.e.,* layers are typically more coarse-grained than functions).

2.1.1 Layer Parallelism

Layer Parallelism is a "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 each incoming and outgoing message may be processed in parallel as they flow through the "layer pipeline" (shown in Figure 1 (1)). The following pseudo-code illustrates the general structure of a process in the Layer Parallelism model:

```
procedure layer_n_service is
begin
   loop
      receive next message from queue
      perform layer processing
      if additional processing is needed then
         pass message either up or down
            to the next process via IPC
   end loop
end
```

An actual implementation generally contains a fixed amount of processes (typically limited by the number of protocol layers) that operate in a "producer/consumer" manner by performing protocol processing and passing messages via interprocess communication (IPC) mechanisms between the layer pipeline stages.

Strict Layer Parallelism is often characterized by potentially high process management and communication overhead (particularly on a uni-processor, due to costs associated

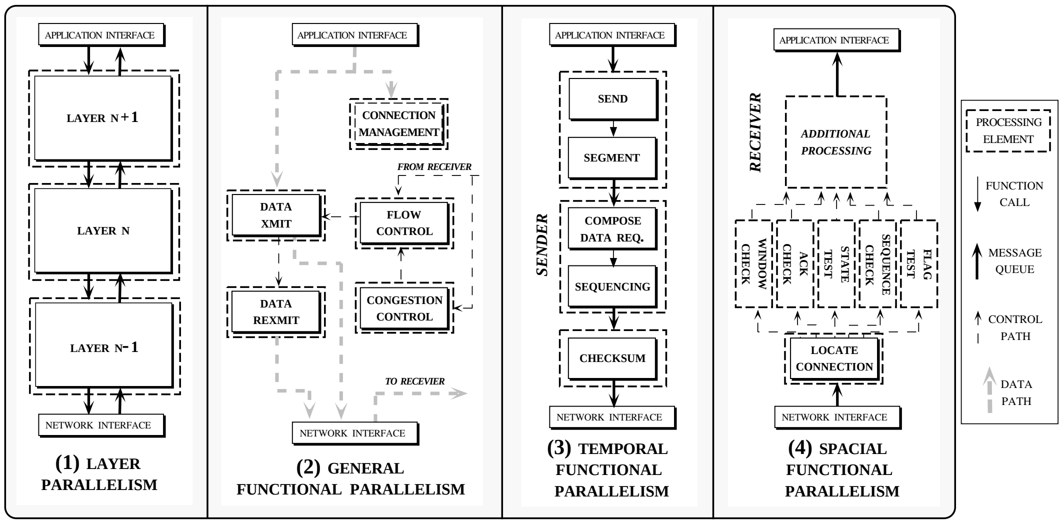


Figure 1: Task-based Process Architectures

with context switching [24]), and minimal support for load balancing (since processes are dedicated to specific protocol layers). Intra-layer buffering, inter-layer flow control, and stage balancing[3] are also typically necessary since processing activities in each layer may not execute at the same rate. Few transport systems utilize pure Layer Parallelism due to the performance overhead, although the XINU TCP/IP implementation [26] uses this approach to simplify the design and implementation of its networking subsystem. In addition, an empirical study of various software architectures for efficiently performing Layer Parallelism is presented in [23].

2.1.2 Functional Parallelism

Functional Parallelism is a "fine-grained" Task-based process architecture that utilizes one or more processes to execute multiple protocol functions (such as header composition, acknowledgement, retransmission, segmentation, reassembly, and routing) in parallel. Figure 1 (2) illustrates the general Functional Parallelism approach, where protocol functions (or clusters of protocol functions) are encapsulated in parallel finite-state machines that communicate by passing control and data messages to each other [27]. This behavior is typically characterized as follows:

```
procedure function_n_service is
begin
   loop
      receive next message from queue
      perform protocol function processing
      if more processing is necessary then
         pass the message to the next
            appropriate process(es) via IPC
   end loop
end
```

Several variants of the general Functional Parallelism model are illustrated in Figure 1 (3) and Figure 1 (4). Figure 1 (3) illustrates a configuration with *temporal* parallelism [25], where several processes cooperate as a pipeline to execute clusters of protocol functions on messages flowing through the sender-side of a protocol session. A process in this configuration is generally structured as follows:

```
procedure temporal_parallelism_service is
begin
   loop
      receive next message from queue
      perform protocol function processing
      if more processing is necessary then
         pass message to next process
            in the pipeline via IPC
   end loop
end
```

Figure 1 (4) illustrates another Functional Parallelism variant involving *spacial* parallelism, where multiple protocol functions (such as retransmission, flow control, congestion control, and presentation conversions) are performed in parallel on fields in each message (the final results may be discarded if errors are detected at intermediate stages). The Horizontally-Oriented Protocol Structure (HOPS) architecture [3] and the Multi-Stream Protocol (MSP) [13] exemplify this latter approach, which may be characterized by the following pseudo-code:

```
procedure spacial_parallelism_service is
begin
   loop
      receive next message from queue
      cobegin
         execute protocol functions 1..n
            on appropriate portions of the
            message
      coend
      if more processing is necessary then
         pass message to next
            process(es) via IPC
   end loop
end
```

The selection of variants such as temporal or spacial paral-

[3]Stage balancing involves clustering protocol functions in order to equalize the time spent at each process in a pipeline [25]. In general, strict adherence to the layer boundaries specified by conventional communication models (such as the ISO OSI reference model) complicates stage balancing.

lelism is highly dependent upon characteristics of the multiprocessor hardware platform and the structure of the communication protocol.

Functional Parallelism is often associated with "delayered" communication models [3, 10] that simplify stage balancing by relaxing conventional layering boundaries and enabling more propitious clustering of protocol functions. Within a cluster of functions executing within a single process, control flow is typically transferred as a consequence of the hardware updating a program counter to reference the next executable protocol function or instruction. Mechanisms for transferring control between processes, on the other hand, depend on the underlying process architecture, operating system, and hardware platform. For instance, IPC mechanisms may be necessary to transfer control between functions in different clusters that are executing on separate processing elements in a non-shared memory platform. Conversely, if several clusters are executing concurrently on separate threads in a shared address space, control may be transferred between functions by simply traversing a pointer link to the next cluster. Depending on the underlying process architecture and hardware platform, synchronization primitives may be necessary to protect resources shared between concurrently executing threads of control.

Performance experiments indicate that Task-based process architectures appear poorly suited for platforms where the number of prototocol task clusters exceeds the number of PEs [23]. This situation results is high levels of context switching overhead incurred during protocol processing. In general, careful protocol design and implementation, along with contention-free memory [12], may be necessary to minimize overhead resulting from communication and synchronization between functions executing in separate processes. For example, sophisticated message management facilities [28] may be required on multi-processor platforms to account for *cache affinity* interactions [29] when exchanging messages between PEs with separate instruction and data caches. Another potential limitation is the fixed amount of available parallelism, which is restricted by the number of layers or functions (this may not be a serious disadvantage if only a small number of PEs are available). On the other hand, Task-based approaches based on pipelining are relatively simple to design and implement. For instance, they typically map directly onto conventional layered communication models using well-structured "producer/consumer" designs where concurrency control within a layer is straightforward [22]. Likewise, Task-based approaches appear well-suited for multi-processor architectures (such as transputers [10]) that lend themselves to efficient pipelined interconnections of multiple PEs.

2.2 Message-based Process Architectures

Message-based process architectures associate processes with connections or messages rather than protocol layers or functions. Two message-based process architectures supported by ADAPTIVE are *Connectional Parallelism* and *Message Parallelism*. The primary difference between these approaches involve the granularity with which messages are demultiplexed onto processes. In particular, Connectional Parallelism demultiplexes all messages bound for the same connection to the same process, where as Message Parallelism typically demultiplexes messages to any suitable process (various scheduling disciplines such as round-robin [12] and adaptive load balancing [30] techniques may be used to determine an approach process).

2.2.1 Connectional Parallelism

Connectional Parallelism is a coarse-grained Message-based process architecture that associates a separate process with every open connection. Figure 2 (1) illustrates this approach, where connections C_1, C_2, C_3, and C_4 are each bound to separate processes that execute the requisite protocol functions on all messages associated with their connection. The following pseudo-code illustrates the general technique:

```
procedure connection_service is
begin
   while connection open loop
      receive next message from queue
      if outgoing message
         call write.put routine
      else if incoming message
         call read.put routine
   end loop
end
```

The general structure of a `put` routine is designed as follows:

```
procedure put is
begin
   perform protocol function processing
   if more processing is necessary then
      pass message to next put
         routine via function call
end
```

Depending on the number of layers and/or functions involved in the protocol processing, there may be several `put` routines invoked to process each message sent to a connection. For the outgoing direction, it relatively easy to determine the appropriate `connection_service` process where a particular message belongs. For the incoming direction, however, a device driver or packet filter typically must perform a demultiplexing operation to determine which process a message is destined for. In general, Connectional Parallelism is well-suited for protocols that demultiplex early in their protocol stack since it is difficult to maintain a strict process-per-connection association across multiplexing boundaries [31, 32].

Connectional Parallelism is relatively simple to implement if an OS allows multiple activities (such as system calls, device interrupts, daemon processes) to proceed in parallel. Moreover, Connectional Parallelism exhibits low communication, synchronization, and process management overhead [14], as long as the number of PEs is greater than or equal to the number of connections. For instance, since all protocol

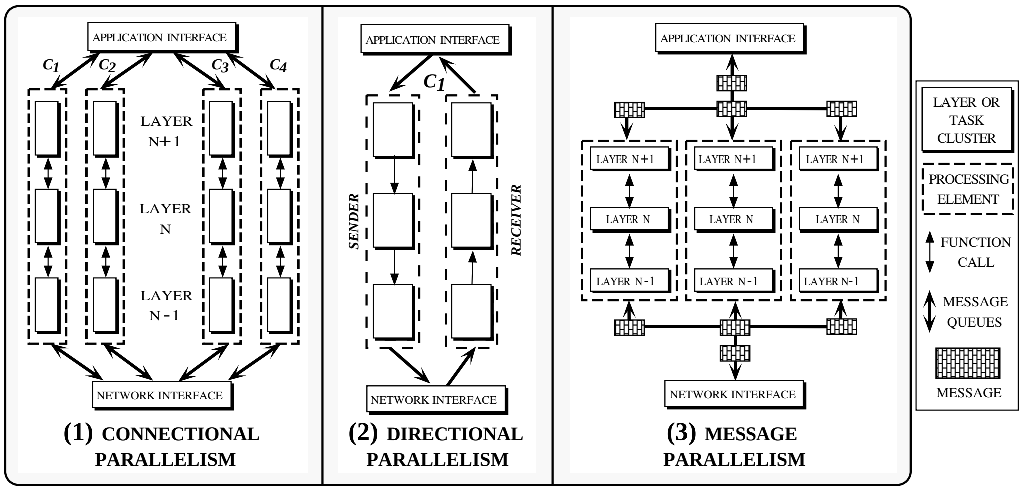


Figure 2: Message-based Process Architectures

context information may be associated with a process, messages may be passed between protocol layers via procedure calls, rather than using more complicated and costly interprocess communication (IPC) mechanisms. However, Connectional Parallelism only utilizes multi-processing to improve *aggregate* end-system performance, *i.e.,* each individual connection still executes sequentially.

Figure 2 (2) illustrates a variation of Connectional Parallelism called Directional Parallelism that associates a separate process with the sender-side and the receiver-side of a single connection [33, 34] in order to improve the utilization of available PEs. The general process architecture structure of this approach is as follows:

```
procedure directional_service is
begin
   while connection open loop
      receive next message from queue
      call put routine
   end loop
end
```

Note that this approach requires some external agent (*e.g.,* a device driver, packet filter, or application interface layer) to separate the incoming and outgoing messages.

In general, the Directional Parallelism model requires a high degree of independence between the sender and receiver portions of a protocol [35], as well as bi-directional flow of application data [27]. A limitation with both the Connectional and Directional process architectures is that load balancing across multiple processes is difficult with Connectional Parallelism since a highly active connection may swamp its process with messages, leaving other processes tied up at less active or idle connections. In addition, the Connectional Parallelism approach only applies to connection-oriented protocols.

2.2.2 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 (3), a process receives a message from an application or network interface and performs most or all of the protocol processing functions on that message. The following pseudo-code characterizes the work performed by each process:

```
procedure message_service is
begin
   loop
      receive next message from queue
      call put routine
   end loop
end
```

The general structure of a `put` routine is designed as follows:

```
procedure put is
begin
   perform protocol function processing
   if more processing is necessary then
      pass message to next put
         routine via function call
end
```

Unlike Connectional Parallelism, the device driver does *not* perform demultiplexing according to connection-related information in a message.

A large degree of potential parallelism exists with this approach, depending on the number of messages exchanged, rather than the number of connections, layers, or functions. Moreover, processing loads may be balanced more evenly among processes since each incoming message may be dispatched to an available process. The primary disadvantages of Message Parallelism involve the overhead resulting from (1) resource management and scheduling support necessary to associate a process with each message, (2) maintaining proper sequencing for events that must not be processed out-of-order [36], (3) synchronization and mutual exclusion operations that also serialize access to resources (such as protocol control blocks that store information such as round-trip time estimates, retransmission queues, and addressing information) shared between messages destined for the same session. Synchronization becomes particularly complicated if

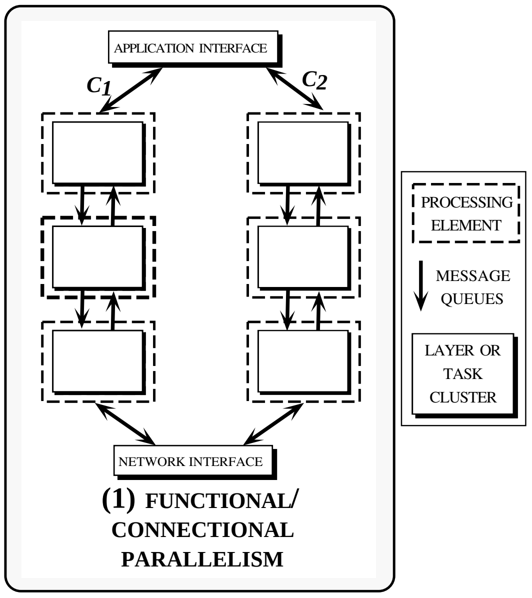


Figure 3: Hybrid Process Architectures

efficient access to shared memory is not available (such as in certain transputer environments). For connection-oriented protocols (such as TCP or TP4), this synchronization overhead may significantly reduce throughput and increase variance in message processing delay [14]. On the other hand, Message Parallelism appears quite suitable for connectionless or request-response protocols, where minimal interdependencies exist between consecutively arriving or departing messages. A number of projects have discussed, simulated, or utilized Message Parallelism as the basis for their process architecture structure [12, 11, 18, 14].

Compared with the Task-based approaches, Message-based process architectures are characterized by more dynamic use of processes, which may enable them to scale-up to utilize larger numbers of PEs effectively. On the other hand, this scalability may be of limited value if a platform possesses a small number of PEs, which is typically the case for modern workstations and PCs. In addition, the increased dynamism also entails more sophisticated resource allocation and management facilities.

2.3 Hybrid Process Architectures

Hybrid process architectures may be formed by combining certain features discussed above. For instance, Functional/Connectional Parallelism (shown in Figure 3 (1)) associates multiple processes with each connection. The general structure of this process architecture is as follows:

```
procedure layer_n_service is
begin
   loop
      get message from queue
      perform protocol function processing
      if more processing is necessary then
         pass message to next process via IPC
end
```

As with Connectional Parallelism, some external agent must demultiplex the messages onto the appropriate process to initiate the protocol processing.

2.4 Process Architecture Performance Factors

The performance of the process architectures described above is influenced by various *external* and *internal* factors (shown in Figure 4). External factors generally treat the transport system as a "black box," and are useful for evaluating end-to-end performance *without* knowledge of, or modification to, a transport system's implementation. External factors that affect process architecture performance include (1) *application characteristics – e.g.,* the number of simultaneously active sessions, the class of service required by applications (such as reliable/non-reliable and real-time/non-real-time [1]), direction of data flow (*i.e.,* uni-directional vs. bi-directional), and the type of traffic generated by applications, (2) *protocol characteristics – e.g.,* the class of protocol (such as connectionless, connection-oriented, and request/response) used to implement application services, and (3) *network characteristics – e.g.,* attributes of the underlying network environment (such as frame-size and channel-speed).

Internal factors, on the other hand, represent hardware- and software-dependent policies and mechanisms that characterize a transport system's implementation such as:

• **Degree of Parallelism:** Message-based process architectures may utilize a large number of processing elements effectively, whereas the Task-based approaches possess limitations that restrict their scalability. Conversely, certain alternatives (such as pipelined Functional Parallelism) do not function efficiently *without* multiple processing elements due to factors such as process management overhead.

• **Process Management Overhead:** process architectures exhibit different context switching and scheduling costs that depend on factors such as the number of available PEs, the type of scheduling policies employed (*e.g.,* preemptive vs. non-preemptive), and the protection domain (*e.g.,* user-mode vs. kernel-mode) in which the protocols execute.

• **Synchronization Overhead:** implementing communication protocols that execute concurrently often requires serializing 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

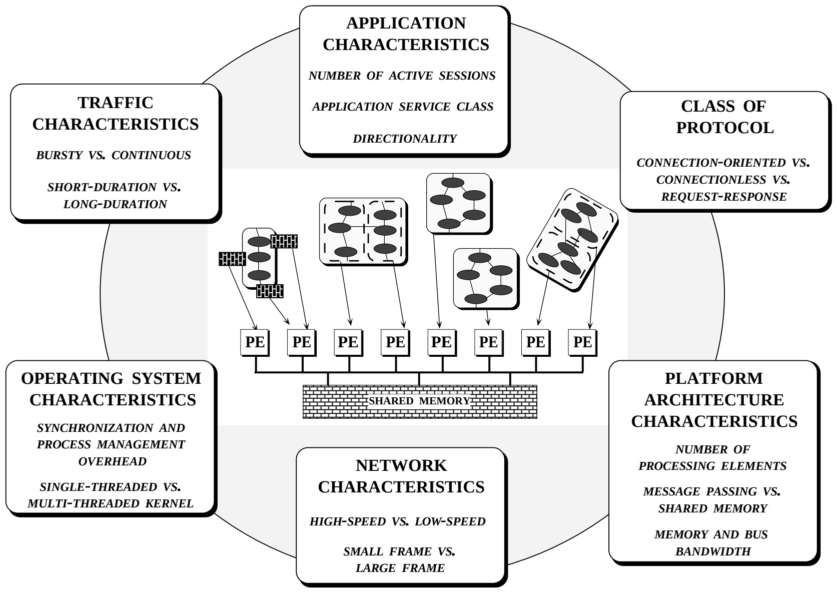

APPLICATION CHARACTERISTICS
NUMBER OF ACTIVE SESSIONS
APPLICATION SERVICE CLASS
DIRECTIONALITY
TRAFFIC CHARACTERISTICS
BURSTY VS. CONTINUOUS
SHORT-DURATION VS. LONG-DURATION
CLASS OF PROTOCOL
CONNECTION-ORIENTED VS. CONNECTIONLESS VS. REQUEST-RESPONSE
PE
SHARED MEMORY
OPERATING SYSTEM CHARACTERISTICS
SYNCHRONIZATION AND PROCESS MANAGEMENT OVERHEAD
SINGLE-THREADED VS. MULTI-THREADED KERNEL
NETWORK CHARACTERISTICS
HIGH-SPEED VS. LOW-SPEED
SMALL FRAME VS. LARGE FRAME
PLATFORM ARCHITECTURE CHARACTERISTICS
NUMBER OF PROCESSING ELEMENTS
MESSAGE PASSING VS. SHARED MEMORY
MEMORY AND BUS BANDWIDTH

Figure 4: External and Internal Factors Influencing Process Architecture Performance

via Message Parallelism) may incur significant synchronization overhead due to the cost of managing locks that serialize access to shared objects [14]. In addition to reducing overall throughput, synchronization bottlenecks resulting from lock contention lead to un-predictable response time that complicates the delivery of constrained-latency multimedia services [37]. Other sources of synchronization overhead involves contention for shared hardware resources such as I/O buses and global memory [38]. In general, hardware contention represents a hard upper limit on the benefits that may be accrued from multi-processing [23].

• **Communication Overhead:** Task-based process architectures generally require IPC to exchange messages between protocol processing components executing on separate PEs. Communication overhead results from memory-to-memory copying, message manipulation operations (such as checksum calculations and compression), and message passing costs incurred from synchronization and process management. In general, techniques for minimizing communication overhead utilize (1) sophisticated buffer management schemes that minimize data copying [39, 40, 41], (2) integrated layer processing techniques [42, 31], and (3) single-copy network/host interface adapters [43].

• **Load Balancing:** certain process architectures (such as Message Parallelism) strive to utilize multiple PEs equitably, whereas others (such as Connectional, Layer, or Functional Parallelism) may underutilize or overutilize the available PEs under various circumstances involving network traffic patterns and application characteristics. Depending on the scheduling policies employed, process architectures that balance PE load generally help to alleviate processing bottlenecks [12].

3 Process Architecture Support in ADAPTIVE

Given the diversity of external and internal factors described in the previous section, it appears unrealistic to expect a single process architecture to be appropriate for all application/OS/network hardware and software configurations. Therefore, ADAPTIVE provides an integrated framework of resources and tools [44] to support the development of, and experimentation with, a variety of process architectures. By controlling for confounding factors (such as platform architecture, operating system, protocol implementation, application requirements, and network characteristics), ADAPTIVE enables precise measurement of how a particular process architecture impacts application and transport system performance. The goal is to identify the circumstances where different process architectures result in significant performance improvements or degradations. This section focuses primarily on ADAPTIVE's process architecture support; other aspects of ADAPTIVE are described in [20, 19, 45].

The primary components of ADAPTIVE are designed to enhance both communication service flexibility and performance. To enhance service flexibility, ADAPTIVE maintains a collection of reusable "building block" protocol functions (such as acknowledgment, retransmission, segmentation, reassembly, sequencing, checksumming, and routing) in a *protocol resource pool* [44]. These functions may be composed together to generate application-tailored *protocol machines* that are customized for different application service classes [1] (shown in Figure 5 (1)). A protocol machine contains a collection of functions that provide the minimal set of functionality required to perform a particular application service (such as transferring voice, video, text, and image data). A protocol machine *configuration* describes the

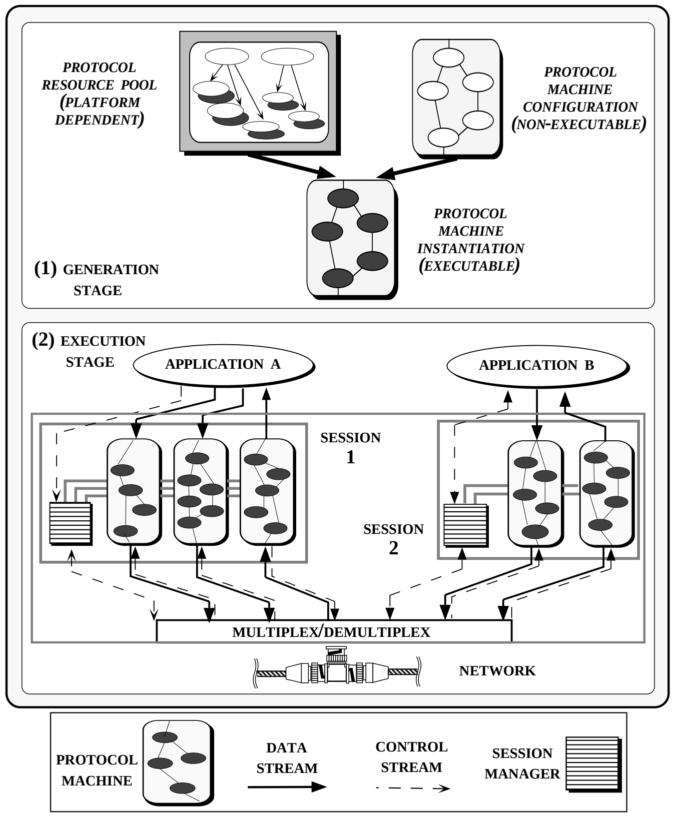


Figure 5: ADAPTIVE Architecture

functional characteristics of a protocol machine, where as a protocol machine *instantiation* is an executable representation containing platform-dependent protocol resources (such as object code and data) that may be optimized to run efficiently on a particular target platform [19].

To support complex applications (such as teleconferencing), multiple protocol machines may be consolidated to form a *session*. A *session manager* coordinates a set of related protocol machines by generating and interpreting session control information and performing various management tasks such as adding, modifying, or deleting data streams dynamically. Figure 5 (2) depicts the relationships between these various entities during execution. In this figure, application A maintains two sessions: session 1 contains two outgoing data streams and one incoming data stream and session 2 contains a single outgoing data stream. Each data stream is implemented by a different protocol machine, which is coordinated by a session manager in each session. ADAPTIVE utilizes a de-layered communication model that demultiplexes messages onto protocol machines as soon as possible (*e.g.,* at the data link or network layer) to (1) reduce or eliminate unnecessary or replicated functionality (such as per-layer demultiplexing and error checking) and (2) increase the amount of parallelism available to the transport system.

To enhance performance, application-tailored protocol machines contain synchronization and mutual exclusion objects necessary to execute efficiently and correctly on multiprocessor platforms. ADAPTIVE is targeted for coarse-grained (*i.e.,* 4 to 30 PEs), symmetric shared memory multiprocessor platforms running general-purpose operating systems (such as UNIX [4], MACH [46], and NT [6]). These operating systems generally utilize a multi-threaded kernel that executes multiple system calls and device interrupts in parallel [5]. To provide a realistic environment for experimentation, ADAPTIVE's protocol implementations execute within the OS kernel, rather than in user-space [47]. Implementing protocols in the kernel helps reduce scheduling, context switching, and protection-domain boundary crossing overhead [48] and often allows more predictable response time due to the use of "wired" (rather than paged) memory.

3.1 Process Architecture Components

To avoid redeveloping device drivers, message buffering schemes, and other transport system infrastructure components, ADAPTIVE is hosted within the STREAMS framework. STREAMS offers a modular and portable set of system calls, data structures, and utility routines that implement bi-directional, kernel-resident character-based I/O subsystems [21]. Contemporary implementations of STREAMS [4, 5, 49, 50] utilize shared memory, symmetric multiprocessing capabilities within a multi-threaded kernel address space.[4]

The STREAMS framework contains the standard components (*e.g.,* STREAM *heads*, *modules*, *multiplexors*, and *drivers*) shown in Figure 6. A STREAM head provides a queueing point where application data is segmented and reassembled into discrete messages that conform to the maximum size of a "transport interface data unit" (which represents the largest message an application may pass to a Stream) [51]. These messages are passed "downstream" from a STREAM head through zero or more modules and/or multiplexers to a driver, where they may be transmitted via a network interface to the appropriate underlying network. Likewise, drivers receive incoming messages from network interface attachments. These messages may be passed "upstream" through modules and/or multiplexors to the appropriate STREAM head. The STREAM head coalesces these messages into buffers provided by the receiver and notifies the recipient application process that data has arrived.

ADAPTIVE uses STREAM module and multiplexor components to cluster protocol functions. A module is linked together with its adjacent upstream and downstream modules via a single pair of *read* and *write* queues. A multiplexor, on the other hand, contains a set of queue pairs that may be linked above and below the multiplexor in essentially arbitrary configurations to form multiplexed internetworking protocol suites. In addition to maintaining a link to its adjacent module, each of the queues in an open Stream may store

[4]In the following discussion, the term "STREAMS" indicates the overall framework, whereas the term "Stream" refers to a cooperating set of data structures and subroutines that link an individual application session in user-space with optional protocol processing components and a network device driver in the kernel.

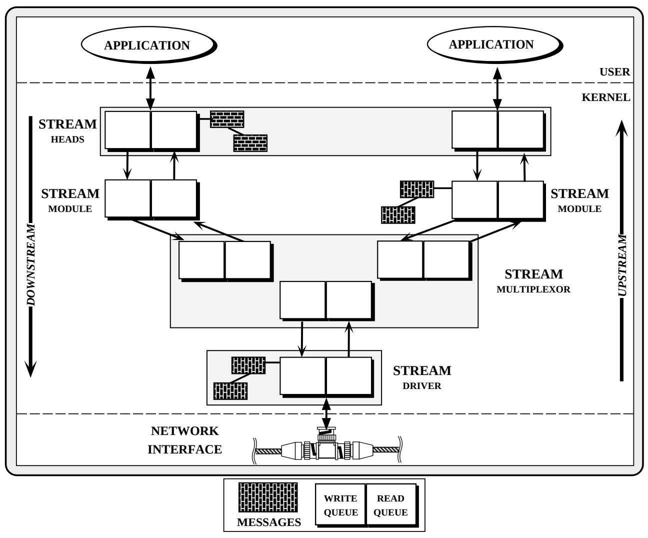


Figure 6: The STREAMS Architecture

a list of control and data messages that are sorted in "priority-order." Read queues store messages arriving from network devices; write queues store messages generated by applications. The overhead of passing messages between modules and multiplexors is minimized by passing pointers to messages rather than copying data as the messages are processed and exchanged between modules and/or multiplexors.

Queues also contain several standard subroutines that perform (1) Stream initialization and termination activities (such as allocating and releasing per-session protocol control blocks, respectively) and (2) *immediate* and/or *deferred* protocol processing functions on messages flowing through a Stream. Immediate processing is performed by a queue's `put` subroutine when a message arrives at a queue. Protocol processing operations that must be invoked immediately (such as handling high-priority TCP "urgent data" messages) are typically performed by `put`. Deferred processing is performed by a queue's `service` subroutine. A `service` subroutine typically implements protocol operations that do not execute in short, fixed amounts of time (such as performing a three-way handshake to establish an end-to-end network connection) or that will block indefinitely due to layer-to-layer flow control conditions within a Stream.

To enhance efficiency, the STREAMS components execute within the OS kernel.[5] To enhance flexibility, the module and multiplexor components may be linked together dynamically by user-level or kernel-level commands to form complete protocol suites (such as those specified by the Internet, ISO OSI, and F-CSS [44] communication models). To enhance modularity and reusability, the connectionless and connection-oriented protocols implemented in ADAPTIVE conform to the Transport Provider Interface (TPI) [53]. The TPI provides a message-based service interface to kernel-resident protocol stacks. This interface shields applications from the implementation details of a particular communication protocol. Applications access the underlying TPI components via a C++ veneer [54] to the Transport Layer Interface (TLI) [55]. This C++ veneer contains classes and operations that provide local context management, connection establishment and termination, data transfer, and option handling.

In general, the modularity of the STREAMS components facilitates experimentation with different process architectures by controlling confounding factors such as platform architecture, operating system, and protocol implementation. For example, the process architecture alternatives described in Section 2 may be implemented via STREAMS by associating processes with different module and multiplexor configurations. As described in the following section, the STREAMS-based version of ADAPTIVE supports Layer Parallelism, Functional Parallelism, Connectional Parallelism, and Message Parallelism, as well as several hybrid process architectures.

4 Implementing Alternative Process Architectures

This section illustrates the implementation of several process architectures via a combination of STREAMS components and multiple processes operating in kernel-mode.[6] These examples are based upon STREAMS implementations of the data transfer and reception portions of a non-proper subset of TCP and UDP. A particular process architecture may be selected implicitly by ADAPTIVE's higher-level tools and/or explicitly by developers [19]. For example, developers typically instrument protocol machines with various synchronization objects (such as mutex and condition variables, counting semaphores, and readers/writer locks [56]). This instrumentation process is facilitated by several features of the C++ language 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 [57]. For example, protocol machines developed with reusable protocol mechanism objects ADAPTIVE's protocol resource pool may be instrumented automatically with synchronization stubs. Depending on the process architecture, these protocol mechanism objects are conditionally compiled to activate the mutual exclusion code required to synchronize interactions between objects at run-time. The intent is to decouple the operations that implement the protocol functions from the operations and synchronization objects required to implement the process architecture framework [14]. When combined with the module and multiplexor STREAMS components, these synchronization objects enable the flexible configuration of protocol machines that execute via one or

[5]Protocol processing functionality may also be migrated to off-board processors [52] due to STREAMS's emphasis on well-defined service interfaces and message passing.

[6]The current implementation uses the kernel-level threads [4] available within the SunOS 5.2 operating system.

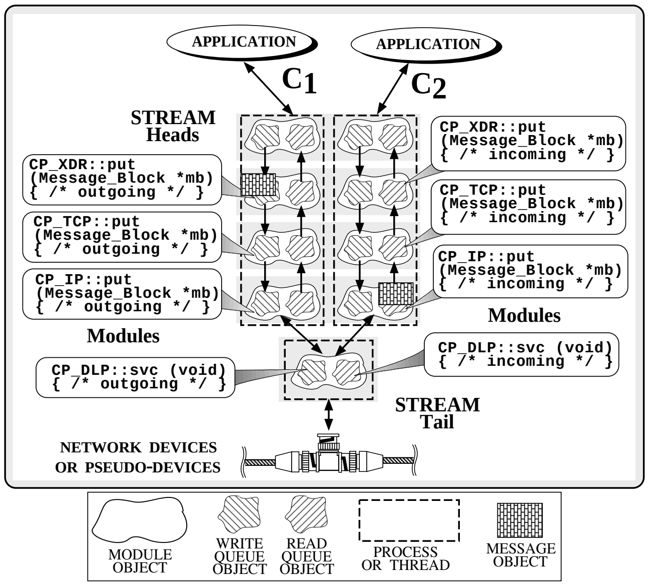

APPLICATION
APPLICATION
C1
C2
STREAM Heads
CP_XDR::put (Message_Block *mb) { /* outgoing */ }
CP_TCP::put (Message_Block *mb) { /* outgoing */ }
CP_IP::put (Message_Block *mb) { /* outgoing */ }
Modules
CP_XDR::put (Message_Block *mb) { /* incoming */ }
CP_TCP::put (Message_Block *mb) { /* incoming */ }
CP_IP::put (Message_Block *mb) { /* incoming */ }
Modules
CP_DLP::svc (void) { /* outgoing */ }
CP_DLP::svc (void) { /* incoming */ }
STREAM Tail
NETWORK DEVICES OR PSEUDO-DEVICES
MODULE OBJECT
WRITE QUEUE OBJECT
READ QUEUE OBJECT
PROCESS OR THREAD
MESSAGE OBJECT

Figure 7: Connectional and Directional Parallelism

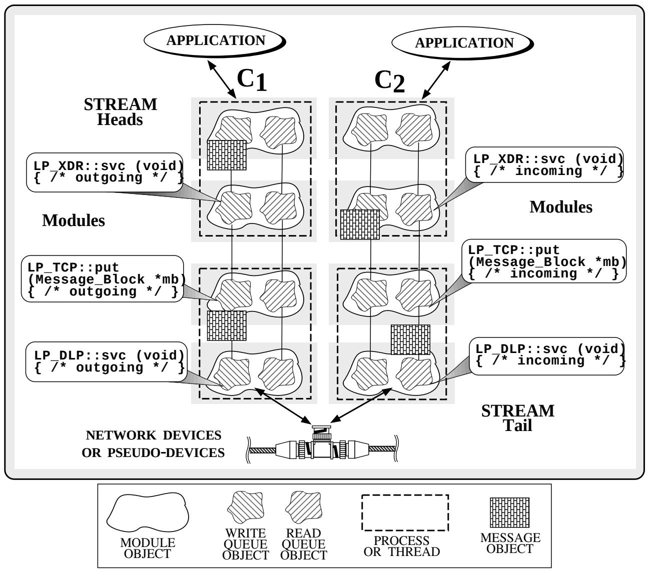

APPLICATION
APPLICATION
C1
C2
STREAM Heads
LP_XDR::svc (void) { /* outgoing */ }
Modules
LP_TCP::put (Message_Block *mb) { /* outgoing */ }
LP_DLP::svc (void) { /* outgoing */ }
LP_XDR::svc (void) { /* incoming */ }
Modules
LP_TCP::put (Message_Block *mb) { /* incoming */ }
LP_DLP::svc (void) { /* incoming */ }
STREAM Tail
NETWORK DEVICES OR PSEUDO-DEVICES
MODULE OBJECT
WRITE QUEUE OBJECT
READ QUEUE OBJECT
PROCESS OR THREAD
MESSAGE OBJECT

Figure 8: Hybrid Functional/Connectional Parallelism

more process architectures with a minimal amount of redevelopment effort.

4.1 Example 1: Connectional and Directional Parallelism

The protocol machines for Connection C_1 and C_2 depicted in Figure 7 illustrate two related variants of the Connectional Parallelism process architecture. Connection C_1 associates a separate process with the protocol machines implemented via its write and read queues. The write queue's `wput` subroutine performs all outgoing protocol processing operations on messages sent from an application before passing the messages to the network interface. Likewise, the read queue `rput` subroutine performs all incoming protocol processing operations on messages received from the network before passing them up to an application. Note that demultiplexing is performed at the network interface, and once a process begins execution all the context information for this protocol session is directly available. This enables a process to operate on its connection's messages without incurring much additional demultiplexing, synchronization, and process management overhead.

ADAPTIVE's modular architecture enables controlled measurement of the performance impact that results from reconfiguring a process architecture. For example, the Connectional Parallelism implementation (shown in Connection C_1) may be modified to use a process architecture that associates separate processes with the sender-side and receiver-side of each connection (shown in Connection C_2). This Directional Parallelism approach utilizes additional parallelism without modifying most of the other STREAMS components and protocol machine functions.

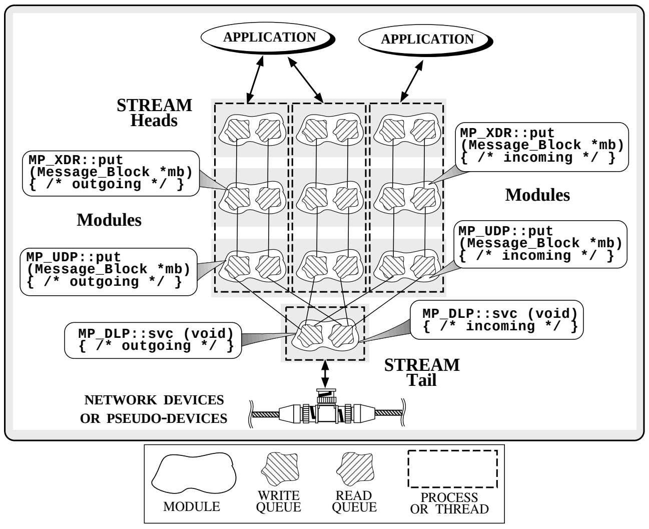
4.2 Example 2: Functional/Connectional Parallelism

The protocol machines illustrated in Figure 8 utilize a hybrid Functional/Connectional Parallelism process architecture that associates separate processes (each executing clusters of protocol functions) with a particular connection. As shown in the figure, processes cooperate in a producer/consumer manner, operating in parallel on the header and data fields of multiple incoming and outgoing messages. A daemon process is associated with every queue, and depending on the "direction" (*i.e.,* incoming or outgoing) of a message, each queues' `wsvc` or `rsvc` subroutine performs certain protocol functions before passing the message to an adjacent queue accessed via a separate process.

Depending on factors such as the ratio of protocol functions (which are typically fixed when a protocol machine is configured) to connections (which may increase or decrease dynamically at run-time), the hybrid Functional/Connectional Parallelism process architecture may utilize a larger amount of available parallelism, compared with the Connectional or Directional Parallelism approaches described above. As with the Connectional Parallelism example, it is possible to modify certain aspects of the protocol machines to utilize an alternative process architecture. For instance, rather than associating processes with clusters of application-tailored protocol functions, the functions may be configured to form a Layer Parallelism process architecture that associates processes and functions with the standard OSI or TCP/IP protocol layers.

4.3 Example 3: Message Parallelism

The preceding examples illustrate STREAMS-based implementations of a connection-oriented TCP-like protocol. Fig-



```
MP_XDR::put
(Message_Block *mb)
{ /* outgoing */ }

MP_UDP::put
(Message_Block *mb)
{ /* outgoing */ }

MP_DLP::svc (void)
{ /* outgoing */ }

MP_XDR::put
(Message_Block *mb)
{ /* incoming */ }

MP_UDP::put
(Message_Block *mb)
{ /* incoming */ }

MP_DLP::svc (void)
{ /* incoming */ }
```

Figure 9: Message Parallelism

ure 9 illustrates a Message Parallelism implementation of a connectionless protocol. The sender and receiver protocol machines shown in Figure 9 are associated with separate daemon processes that handle each arriving or departing message concurrently and independently. The `wput` and `rput` subroutines implement a lightweight UDP-like connectionless protocol that delivers messages up to applications or down to networks without attempting to preserve inter-message ordering.

5 Concluding Remarks

ADAPTIVE provides a framework for developing and experimenting with alternative process architectures to help improve protocol performance, reduce operating system overhead, and simplify transport system development. To support accurate and realistic experiments with alternative process architectures, ADAPTIVE is designed to control for many confounding transport system factors such as communication protocol, operating system, and hardware platform. To facilitate this, ADAPTIVE utilizes a modular architecture that allows developers to hold certain transport system components constant while varying process architecture components and measuring the resulting performance impacts.

To experiment with alternative process architectures, we are developing a prototype implementation of ADAPTIVE that is written in C++ and hosted in the STREAMS framework on a multi-processor UNIX platform [4]. This multiprocessing STREAMS platform supports the development of several different process architectures including Functional Parallelism, Connectional Parallelism, and Message Parallelism. We are currently using ADAPTIVE to implement and evaluate a number of protocol machines that are customized for several classes of multimedia applications (such as audio, video, text, and image data) running on several different networks (such as Ethernet, FDDI, and ATM).

References

[1] 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.

[2] J. S. Turner, "Why We Need Gigabit Networks," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, IEEE, Apr. 1989.

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

[4] 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.

[5] A. Garg, "Parallel STREAMS: a Multi-Process Implementation," in *Proceedings of the Winter USENIX Conference*, (Washington, D.C.), Jan. 1990.

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

[7] 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, Aug. 1987.

[8] A. McRae, "Hardware Profiling of Kernels," in *USENIX Winter Conference*, (San Diego, CA), USENIX Association, Jan. 1993.

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

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

[11] M. Ito, L. Takeuchi, and G. Neufeld, "A Multiprocessing Approach for Meeting the Processing Requirements for OSI," *IEEE Journal on Selected Areas in Communications*, pp. 220–227, Feb. 1993.

[12] 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.

[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 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.

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

[17] D. C. Feldmeier, "Multiplexing Issues in Communications System Design," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 209–219, ACM, Sept. 1990.

[18] 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, Jan. 1991.

[19] D. C. Schmidt and T. 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), pp. 103–118, IFIP/IEEE, Aug. 1994.

[20] 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.

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

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

[23] C. M. Woodside and R. G. Franks, "Alternative Software Architectures for Parallel Protocol Execution with Synchronous IPC," *IEEE/ACM Transactions on Networking*, vol. 1, Apr. 1993.

[24] D. Batory and S. W. O'Malley, "The Design and Implementation of Hierarchical Software Systems Using Reusable Components," *ACM Transactions on Software Engineering and Methodology*, vol. 1, pp. 355–398, Oct. 1992.

[25] 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.

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

[27] T. Braun and M. Zitterbart, "Parallel Transport System Design," in *Proceedings of the 4th IFIP Conference on High Performance Networking*, (Belgium), IFIP, 1993.

[28] B. Lindgren, B. Krupczak, M. Ammar, and K. 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), IEEE, Sept. 1993.

[29] R. Vaswani and J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors," in *Proceedings of the 13th Symposium on Operating System Principles*, (Pacific Grove, CA), pp. 26–40, ACM, Oct. 1991.

[30] H. Massalin and C. Pu, "Fine-grain adaptive scheduling using feedback," *Computing Systems*, vol. 3, pp. 139–173, Winter 1990. Special Issue on selected papers from the Workshop on Experiences in Building Distributed Systems, Florida, October 1989.

[31] M. B. Abbott and L. L. Peterson, "A language-based approach to protocol implementation," *IEEE Journal of Transactions on Networking*, vol. 1, Feb. 1993.

[32] D. L. Tennenhouse, "Layered Multiplexing Considered Harmful," in *Proceedings of the 1st International Workshop on High-Speed Networks*, May 1989.

[33] 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*, pp. 165–180, May 1989.

[34] M. H. Nguyen and M. Schwartz, "Reducing the Complexities of TCP for a High-Speed Networking Environment," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (San Francisco, California), IEEE, 1993.

[35] A. N. Netravali, W. D. Roome, and K. Sabnani, "Design and Implementation of a High Speed Transport Protocol," *IEEE Transactions on Communications*, 1990.

[36] M. Goldberg, G. Neufeld, and M. Ito, "A Parallel Approach to OSI Connection-Oriented Protocols," in *Proceedings of the 3rd IFIP Workshop on Protocols for High-Speed Networks*, (Stockholm, Sweden), May 1992.

[37] R. Govindan and D. P. Anderson, "Scheduling and IPC Mechanisms for Continuous Media," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 68–80, Oct. 1991.

[38] 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.

[39] N. C. Hutchinson, S. Mishra, L. L. Peterson, and V. T. Thomas, "Tools for Implementing Network Protocols," *Software Practice and Experience*, vol. 19, pp. 895–916, Sept. 1989.

[40] C. M. Woodside and J. R. Montealegre, "The Effect of Buffering Strategies on Protocol Execution Performance," *IEEE Transactions on Communications*, vol. 37, pp. 545–554, June 1989.

[41] X. Zhang and A. Seneviratne, "An Efficient Implementation of High-Speed Protocol without Data Copying," in *Proceedings of the 15th Conference on Local Computer Networks*, (Minneapolis, MN), pp. 443–450, IEEE, Oct. 1990.

[42] 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.

[43] G. Watson, D. Banks, C. Calamvokis, C. Dalton, A. Edwards, and J. Lumley, "Afterburner," *IEEE Network Magazine*, vol. 7, July 1993.

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

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

[46] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tavanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," in *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, (Atlanta, GA), pp. 93–112, June 1986.

[47] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska, "Implementing Network Protocols at User Level," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (San Francisco, California), ACM, 1993.

[48] 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 11th Symposium on Operating System Principles (SOSP)*, Nov. 1987.

[49] D. Presotto, "Multiprocessor Streams for Plan 9," in *Proceedings of the United Kingdom UNIX User Group Summer Proceedings*, (London, England), Jan. 1993.

[50] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan, "Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 85–106, Jan. 1993.

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

[52] H. Kanakia and D. R. Cheriton, "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Stanford, CA), pp. 175–187, ACM, Aug. 1988.

[53] OSI Special Interest Group, *Transport Provider Interface Specification*, Dec. 1992.

[54] D. C. Schmidt and T. 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*, (Pittsburgh, PA), pp. 190–201, IEEE, Mar. 1994.

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

[56] A. D. Birrell, "An Introduction to Programming with Threads," Tech. Rep. SRC-035, Digital Equipment Corporation, Jan. 1989.

[57] B. Stroustrup, *The C++ Programming Language, 2nd Edition*. Addison-Wesley, 1991.