Transport System Architectures for High-Performance Communications Subsystems

Douglas C. Schmidt and Tatsuya Suda
Department of Information and Computer Science,
University of California, Irvine
Irvine, CA 92717, U.S.A.

An earlier version of this paper appeared in the IEEE Journal on Selected Areas in Communication, Vol. 11, No. 4, May 1993.

Abstract

Providing end-to-end gigabit communication support for bandwidth-intensive distributed applications requires high-performance transport systems. This paper describes and classifies transport system mechanisms that integrate operating system resources (such as CPU(s), virtual memory, and network adapters) together with communication protocols (such as TCP/IP and XTP) to support applications running on local and wide area networks. A taxonomy is presented that compares and evaluates four widely available transport systems in terms of their support for protocol processing. The systems covered in this paper include System V UNIX STREAMS, the BSD UNIX networking subsystem, the x-kernel, and the Conduit framework from the Choices operating system. This paper is intended to help researchers navigate through the transport system design space by describing alternative mechanisms for developing transport systems.

1 Introduction

The demand for many types of distributed applications is expanding rapidly, and application requirements and usage patterns are undergoing significant changes. When coupled with the increased channel speeds and services offered by high-performance networks, these changes make it difficult for existing transport systems to process application data at network channel speeds.[1] This paper examines transport system mechanisms that support bandwidth-intensive multimedia applications such as medical imaging, scientific visualization, full-motion video, and tele-conferencing. These applications possess quality-of-service requirements that are significantly different from conventional data-oriented applications such as remote login, email, and file transfer.

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

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

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

[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.

primary and secondary storage, and network adapters) and software components (*e.g.,* virtual memory, process architectures, message managers, and protocol graphs) to support the execution of distributed applications.

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

In general, sources of transport system overhead are not decreasing as rapidly as network channel speeds are increasing. This results from factors such as improperly layered transport system architectures [1, 2]. It is also exacerbated by the widespread use of operating systems that are not well-suited to asynchronous, interrupt-driven network communication. For example, many network adapters generate interrupts for every transmitted and received packet, which increases the number of CPU context switches [3, 4]. Despite increasing total MIPS, RISC-based computer architectures exhibiting high context switching overhead that penalizes interrupt-driven network communication. This overhead results from the cost of flushing pipelines, invalidating CPU instruction/data caches and virtual memory translation-lookaside buffers, and managing register windows [5].

Alleviating the throughput preservation problem and providing very high data rates to applications requires the modification of conventional transport system architectures [6]. To help system researchers navigate through the transport system design space, this paper presents a taxonomy of six key transport system mechanisms including the process architecture, virtual remapping, and event management dimensions, as well as the message management, multiplexing and demultiplexing, and layer-to-layer flow control dimensions. The taxonomy is used to compare and contrast four general-purpose commercial and experimental transport systems (System V STREAMS [7], the BSD UNIX network subsystem [8], the *x*-kernel [2], and the Conduit framework from the Choices operating system [9]). The intent of the paper is to explore transport system design alternatives that support distributed applications effectively.

The paper is organized as follows: Section 2 outlines the general architectural components in a transport system; Section 3 describes a taxonomy for classifying transport systems according to their kernel and protocol family architecture dimensions; Section 4 provides a comparison of four representative transport systems; and Section 5 presents concluding remarks.

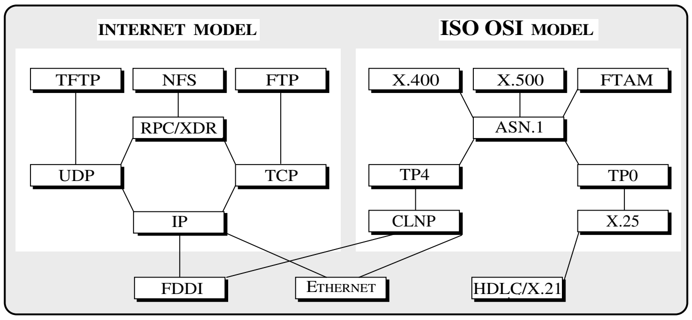

Figure 1: Protocol Graph for Internet and OSI Communication Models

2 Levels of Abstraction in a Transport System Architecture

Transport system architectures provide a framework for implementing end-to-end protocols that support distributed applications operating over local and wide area networks. This framework integrates hardware resources and software components used to implement *protocol graphs* [10]. A protocol graph characterizes hierarchical relations between protocols in communication models such as the Internet, OSI, XNS, and SNA. Figure 1 depicts protocol graphs for the Internet and OSI communication models. Each node in a protocol graph represents a protocol such as RPC/XDR, TCP, IP, TP4, or CLNP.

Protocol graphs are implemented via mechanisms provided by the transport system architecture. Transport systems may be modeled as nested virtual machines that constitute different levels of abstraction. Each level of virtual machine is characterized by the mechanisms it exports to the surrounding levels. The model depicted in Figure 2 represents an abstraction of hardware and software mechanisms found in conventional transport systems. Although certain transport systems bypass or combine adjacent levels for performance reasons [11, 12], Figure 2 provides a concise model of the relationships between major transport system components.

The hierarchical relationships illustrated by the protocol graph in Figure 1 are generally orthogonal to the levels of abstraction depicted by the transport system virtual machines shown in Figure 2. In particular, protocol graphs in Figure 1 are implemented via the transport system mechanisms shown in Figure 2. The following paragraphs summarize the key levels in the transport system, which consist of the *application interface*, *session architecture*, *protocol family architecture*, and *kernel architecture*.

As shown by the shaded portions of Figure 2, this paper focuses on the kernel architecture (described in Section 3.1) and the protocol family architecture (described in Section 3.2). A thorough discussion of the application interface is beyond the scope of this paper and topics involving the session architecture are discussed further in [13]. These two components are briefly outlined below for completeness and to provide a context for discussing the other levels.

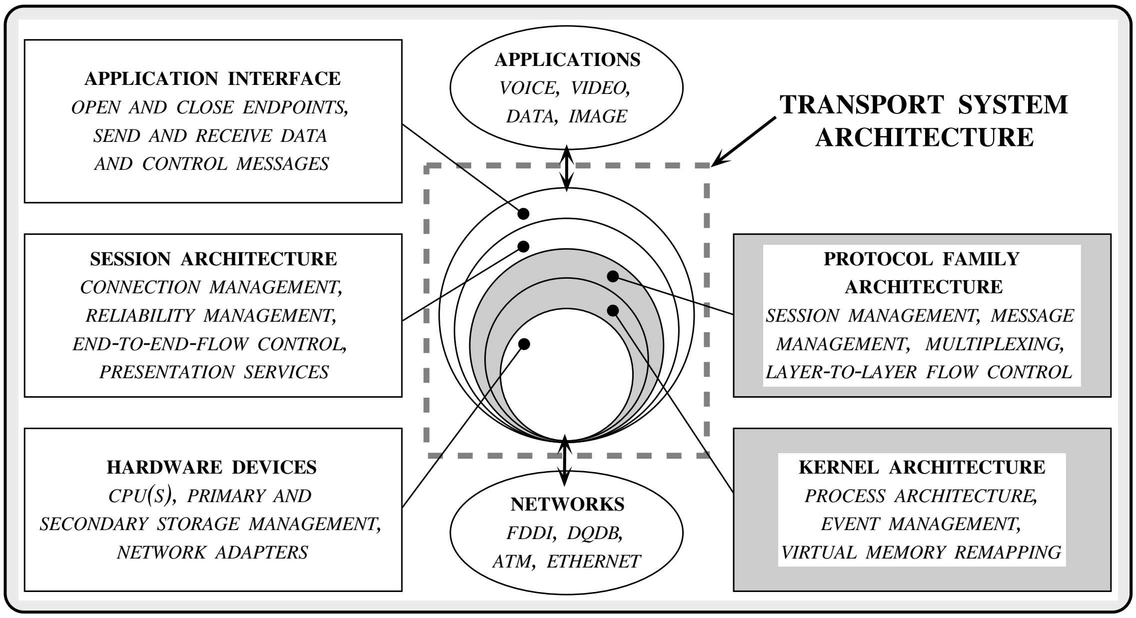

Figure 2: Abstract Architecture for a Transport System

2.1 Application Interface

The *application interface* is the outermost-level of a transport system. Since protocol software often resides within the protected address space of an operating system kernel, programs utilize this application interface to interact with inner-level transport system mechanisms. The application interface transfers data and control information between user processes and the session architecture mechanisms that perform connection management, option negotiation, data transmission control, and error protection. BSD UNIX sockets [8] and System V UNIX TLI [14] are widely available examples of application interfaces.

Performance measurements indicate that conventional application interfaces constitute 30 to 40 percent of the overall transport system overhead [2, 15]. Much of this overhead results from the memory-to-memory copying and process synchronization that occurs between application programs and the inner-level transport system mechanisms. The functionality and performance of several application interfaces is evaluated in [16, 17].

2.2 Session Architecture

The next level of the transport system is the *session architecture*, which performs "end-to-end" network tasks. Session architecture mechanisms are associated with local endpoints of network communication, often referred to as protocol *sessions*.[2] A session consists of data structures that store context information and subroutines that implement the end-to-end protocol state machine operations.

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

Session architecture mechanisms help satisfy end-to-end application quality-of-service requirements involving throughput, latency, and reliability [18]. In particular, quality-of-service is affected by session architecture mechanisms that manage *connections* (*e.g.,* opening and closing end-to-end network connections, and reporting and updating connection context information), *reliability* (*e.g.,* computing checksums, detecting mis-sequenced or duplicated messages, and performing acknowledgments and retransmissions), and *end-to-end flow and congestion* (*e.g.,* advertizing available window sizes and tracking round-trip packet delays). In addition, session architecture mechanisms also manage per-connection *protocol interpreters* (*e.g.,* controlling transitions in a transport protocol's state machine) and *presentation services* (*e.g.,* encryption, compression, and network byte-ordering conversions). Various session architecture issues are examined in [19, 20, 21, 13].

2.3 Protocol Family Architecture

The *protocol family architecture*[3] provides *intra-* and *inter-protocol* mechanisms that operate within and between nodes in a protocol graph, respectively. Intra-protocol mechanisms manage the creation and destruction of sessions that are managed by the session architecture described above. Inter-protocol mechanisms provide message management, multiplexing and demultiplexing, and layer-to-layer flow control.

The primary difference between the session architecture and the protocol family architecture is that session architec-

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

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

Table 1: Transport System Taxonomy Template

ture mechanisms manage the *end-to-end* processing activities for network connections, whereas protocol family architecture mechanisms manage the *layer-to-layer* processing activities that occur within multi-layer protocol graphs. In some cases, these activities are entirely different (*e.g.,* the presentation services provided by the session architecture such as encryption, compression, and network byte-ordering are unnecessary in the protocol family architecture). In other cases, different mechanisms are used to implement the same abstract task.

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

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

2.4 Kernel Architecture

The *kernel architecture*[4] provides mechanisms that manage hardware resources such as CPU(s), primary and secondary storage, and various I/O devices and network adapters. These mechanisms support concurrent execution of multiple protocol tasks on uni- and multi-processors, virtual memory management, and event handling. It is crucial to implement kernel architecture mechanisms efficiently since the application interface and session and protocol family architectures ultimately operate by using these mechanisms. The primary distinction between the protocol family architecture and the kernel architecture is that kernel mechanisms are also utilized by user applications and other OS subsystems such as the graphical user interface or file subsystems. In contrast, protocol family architecture mechanisms are concerned primarily with the communication subsystem.

3 A Taxonomy of Transport System Architecture Mechanisms

Table 1 presents a taxonomy of six key kernel architecture and protocol family architecture mechanisms that support the layer-to-layer computing requirements of protocol graphs end systems. The following section describes the transport system mechanisms presented in Table 1.

3.1 Kernel Architecture Dimensions

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

3.1.1 The Process Architecture Dimension

A process is a collection of resources (such as file descriptors, signal handlers, a run-time stack, etc.) that may support

[4]The term "kernel architecture" is used within this paper to identify mechanisms that form the "nucleus" of the transport system. However, protocol and session architecture components may reside within an OS kernel (BSD UNIX [8], and System V UNIX [7]), in user-space (Mach [22] and the Conduit [9]), in either location (the *x*-kernel [2]), or in off-board processors (Nectar [23] and VMP [4]).

the execution of instructions within an address space. This address space may be shared with other processes. Other terms (such as threads [24]) are often used to denote the same basic concept. Our use of the term process is consistent with the definition adopted in [25].

A process architecture represents a binding between various units of communication protocol processing (such as layers, functions, connections, and messages) and various structural configurations of processes. The process architecture selected for a transport system is one of several factors (along with protocol designs/implementations and bus, memory, and network interface characteristics) that impact overall application performance. In addition, the choice of process architecture also influences demultiplexing strategies [26] and protocol programming techniques [2, 27].

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

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

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

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

  - Context switching between LWPs is less time consuming than HWPs since there is less context information to store and retrieve
  - It may not be necessary to perform a "mode switch" between kernel- and user-mode when scheduling and executing an LWP [28]
  - Communication between LWPs may use shared memory rather than message passing

Note that LWPs may be implemented in kernel-space, user-space, or some hybrid configuration [29].

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

Executing protocol and session mechanisms via multiple processes is often less complicated and error-prone than synchronizing and scheduling these mechanisms manually via coroutines. In addition, coroutines support only interleaved process execution, which limits the benefits of multiprocessing since only one process may run at any given time. In general, it appears that LWPs are a more appropriate mechanism for implementing process architectures than HWPs since minimizing context switching overhead is essential for high-performance [2]. Even with LWPs, however, to it is still important to perform concurrent processing efficiently to reduce the overhead from (1) preempting, rescheduling, and synchronizing executing processes and (2) serializing access to shared resources must be minimized.

(2) Process Architecture Alternatives: Three primary process architecture components in a communication subsystem include (1) the *processing elements* (CPUs), which are the underlying execution agents for both protocol and application code, (2) *data and control messages*, which are typically sent and received from one or more applications and network devices, and (3) *protocol processing tasks*, which perform protocol-related functions upon messages as they arrive and depart. Based upon this classification, two basic categories of process architecture may be distinguished: *task-based* and *message-based*. Each category is characterized by alternative methods for structuring and composing the three communication subsystem components outlined above. In general, task-based process architectures structure multiple CPUs according to units of protocol functionality. Conversely, message-based process architectures structure the CPUs according to the protocol control and data messages received from applications and network interfaces.

In terms of functionality, protocol suites (such as the Internet and ISO OSI reference models) may be implemented using either task-based or message-based process architectures. However, each category of process architecture exhibits different structural and performance characteristics. The structural characteristics differ according to (1) the granularity of the unit(s) of protocol processing (*e.g.,* layer or function vs. connection or message) that execute in parallel, (2) the degree of CPU scalability (*i.e.,* the ability to effectively use only a fixed number of CPUs vs. a dynamically scalable amount), (3) task invocation semantics (*e.g.,* synchronous vs. asynchronous execution) and (4) the effort required to design and

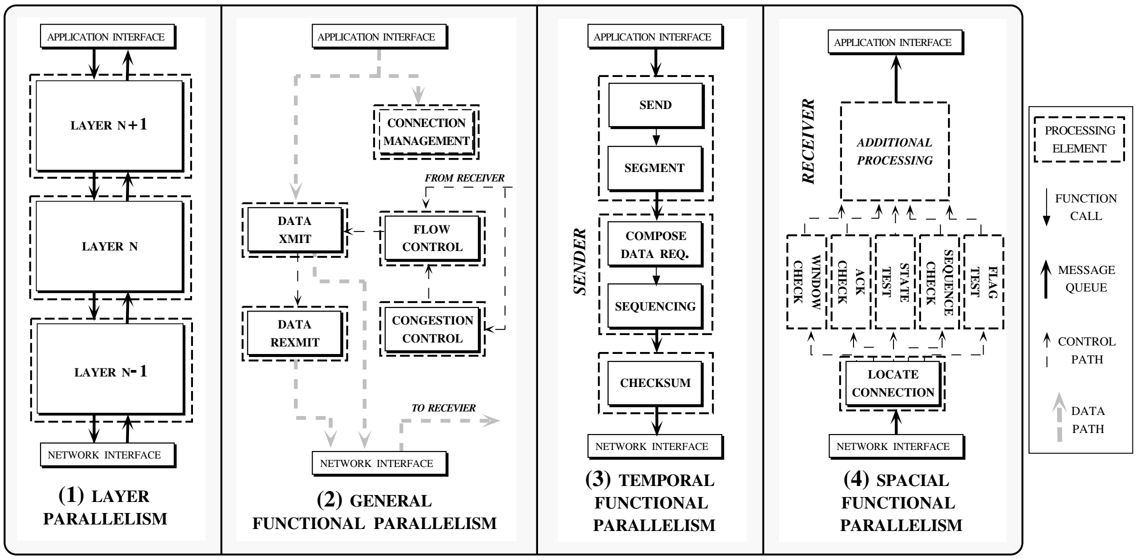

Figure 3: Task-based Process Architectures

implement conventional and experimental protocols and services via a particular process architecture [27]. In addition, different configurations of application requirements, operating system (OS) and hardware platforms, and network characteristics interact with the structural characteristics of process architectures to yield significantly different performance results. For instance, on certain general-purpose OS platforms (such as the System V STREAMS framework on multiprocessor versions of UNIX), fine-grained task-based parallelism results in prohibitively high levels of synchronization overhead [30]. Likewise, asynchronous, rendezvous-based task invocation semantics often result in high data movement and context switching overhead [31].

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

• **Task-based Process Architectures:** Task-based process architectures associate OS processes with protocol layers or protocol functions. Two common task-based process architectures are *Layer Parallelism* and *Functional Parallelism*. The primary difference between these two models involves the *granularity* of the protocol processing tasks. In general, layers are more "coarse-grained" than functions since they cluster multiple protocol tasks together to form a composite service.

- *Layer Parallelism* – Layer Parallelism is a relatively coarse-grained task-based process architecture that associates a separate process with each layer (*e.g.,* the presentation, transport, and network layers) in a protocol stack. Certain protocol header and data fields in outgoing and incoming messages may be processed in parallel as they flow through the "layer pipeline" (shown in Figure 3 (1)). Intra-layer buffering, inter-layer flow control, and stage balancing are generally necessary since processing activities in each layer may execute at different rates. In general, strict adherence to the layer boundaries specified by conventional communication models (such as the ISO OSI reference model) complicates stage balancing.

  An empirical study of the performance characteristics of several software architectures for implementing Layer Parallelism is presented in [31]. Likewise, the XINU TCP/IP implementation [32] uses a variant of this approach to simplify the design and implementation of its communication subsystem.

- *Functional Parallelism* – Functional Parallelism is a more fine-grained task-based process architecture that applies one or more processes to execute protocol functions (such as header composition, acknowledgement, retransmission, segmentation, reassembly, and routing) in parallel. Figure 3 (2) illustrates a typical Functional Parallelism design [33], where protocol functions are encapsulated within parallel finite-state machines that communicate by passing control and data messages to each other. Functional Parallelism is often associated with "de-layered" communication models [3, 34] that simplify stage balancing by relaxing conventional layering boundaries in order to minimize queueing delays and "pipeline stalls" within a protocol stack.

  Several variants of Functional Parallelism are illustrated in Figure 3 (3) and Figure 3 (4). Figure 3 (3) illustrates a *temporal* parallelism configuration [34], where several cooperating processes are pipelined to execute clusters of protocol functions on messages flowing through the sender-side of a connection. Figure 3 (4) illustrates another variant known as *spacial* parallelism, where multiple protocol functions (such as retransmission, flow control, congestion control, and presentation layer 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) [35] exemplify this latter approach.

Implementing pipelined, task-based process architectures is relatively straight-forward since they typically map onto conventional layered communication models using well-structured "producer/consumer" designs [27]. Moreover, minimal concurrency control mechanisms are necessary *within* a layer or function since multi-processing is typically serialized at a service access point (such as the transport or application layer interface). However, performance experiments [31] indicate that task-based process architectures are susceptible to high process management and communication overhead. This becomes particularly problematic if the number of protocol tasks exceeds the number of CPUs, due to the context switching and rescheduling operations performed when transfering messages between protocol tasks. Moreover, task-based variants provide minimal support for load balancing since processes are dedicated to specific protocol layers or functions. In addition, the effectiveness of temporal or spacial parallelism is highly dependent upon characteristics of the multi-processor hardware platform (such as the presence of high-speed I/O interconnection hardware [33] and/or lack of rapid access to shared memory) and the network protocols (such as the capability to process mis-ordered data [36]).

- **Message-based Process Architectures:** Message-based process architectures associate processes with connections or messages rather than protocol layers or functions. Two common message-based process architectures are *Connectional Parallelism* and *Message Parallelism*. The primary difference between these approaches involve (1) the class of protocols that may be supported (*e.g.,* Connectional Parallelism does not directly apply to connectionless protocols) and (2) the granularity at which messages are demultiplexed onto processes. For example, Connectional Parallelism typically demultiplexes all messages bound for the same connection onto the same process, whereas Message Parallelism may demultiplex messages onto any suitable process. In general, various scheduling disciplines such as round-robin [5], adaptive load balancing, and cache affinity [37] preserving techniques may be used when selecting a suitable process.

- *Connectional Parallelism* – Connectional Parallelism is a relatively coarse-grained message-based process architecture that associates a separate process with every open connection. Figure 4 (1) illustrates this approach, where connections C_1, C_2, C_3, and C_4 execute in separate processes that perform the requisite protocol functions on all messages associated with their connection. Within a connection, multiple protocol processing functions are invoked serially on each message as it flows

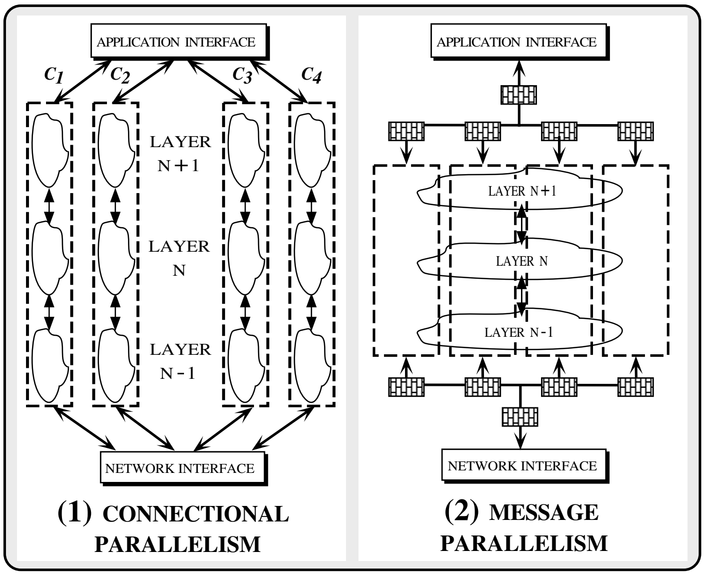

Figure 4: Message-based Process Architectures

through a protocol stack. Outgoing messages typically borrow the thread of control from the application process and use it to shepard one or more messages down a protocol stack [38]. For incoming messages, a device driver or packet filter [39] typically performs demultiplexing operations to determine the correct process for each message. In general, Connectional Parallelism is well-suited for protocols that demultiplex early in their protocol stack since it is difficult to maintain a strict process-per-connection association across demultiplexing boundaries [26].

Connectional Parallelism is relatively simple to implement if an OS allows multiple independent system calls, device interrupts, and daemon processes to operate in parallel [38]. Moreover, if the number of CPUs is greater than or equal to the number of active connections, Connectional Parallelism also exhibits low communication, synchronization, and process management overhead [30] since all connection context information is localized within a particular process address space. This localization is beneficial since (1) pointers to messages may be passed between protocol layers via simple procedure calls (rather than using more complicated and costly interprocess communication mechanisms) and (2) cache affinity properties may be preserved since messages are processed largely within a single CPU cache. The primary limitation of Connectional Parallelism is that it only utilizes multi-processing to improve *aggregate* end-system performance since each individual connection still executes sequentially.

Figure 4 (2) illustrates a variation called Directional Parallelism that associates a separate process with the sender-side and the receiver-side of a single connection [40] in order to improve the utilization of available

CPUs. To be most effective, Directional Parallelism requires a high degree of independence between the sender and receiver portions of a protocol, as well as a bi-directional flow of application control and data messages [33]. In general, load balancing across multiple processes is difficult with both Connectional and Directional Parallelism since highly active connections may swamp their processes with messages, leaving other processing resources tied up at less active or idle connections.

- *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 4 (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. As with Connectional Parallelism, outgoing messages typically borrow the thread of control from the application that initiated the message transfer. A number of projects have discussed, simulated, or utilized Message Parallelism as the basis for their process architecture [5, 41, 2, 42, 25].

  Performance experiments [42] indicate that Message Parallelism scales quite well for connectionless protocols that possess minimal interdependencies between consecutively arriving or departing messages. Moreover, processing loads may be balanced more evenly among processes since each incoming message may be dispatched to an available CPU. The primary disadvantages of Message Parallelism involve overhead resulting from (1) resource management and scheduling support necessary to associate a process with each message, (2) maintaining proper sequencing for messages that must be processed in-order [41, 36], and (3) serializing 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 connection. For connection-oriented protocols (such as TCP or TP4), this synchronization overhead may significantly limit speedups obtainable from multiple CPUs [42] and increase variance in message processing delay.

Compared with task-based approaches, message-based process architectures are characterized by the potential for more dynamic process utilization. In general, a large degree of potential parallelism exists with these approachs, depending on dynamic characteristics (such as messages or connections), rather than on relatively static characteristics (such as the number of layers or protocol functions). Depending on other communication subsystem factors such as memory and bus bandwidth [43], this dynamism may enable message-based process architectures to scale up to a larger number of CPUs. On the other hand, scalability may be of limited value if a platform possesses a small number of CPUs, which is typically the case for modern workstations and high-end PCs. In addition, the increased dynamism of message-based process architectures also entails more sophisticated, and potentially less efficient, resource allocation and process management facilities. For example, a Message Parallelism-based process architecture may require more elaborate OS scheduling mechanisms.

Process Architecture Performance Factors: The performance of the process architectures described above is influenced by various *external* and *internal* factors. External factors include (1) *application characteristics – e.g.,* the number of simultaneously active connections, the class of service required by applications (such as reliable/non-reliable and real-time/non-real-time), the direction of data flow (*i.e.,* uni-directional vs. bi-directional), and the type of traffic generated by applications (*e.g.,* bursty vs. continuous), (2) *protocol characteristics – e.g.,* the class of protocol (such as connectionless, connection-oriented, and request/response) used to implement application and communication subsystem services, and (3) *network characteristics – e.g.,* attributes of the underlying network environment (such as the delivery of mis-ordered data due to multipath routing [36]). Internal factors, on the other hand, represent hardware- and software-dependent communication subsystem implementation characteristics such as:

- *Process Management Overhead* – Process architectures exhibit different context switching and scheduling costs related to (1) the type of scheduling policies employed (*e.g.,* preemptive vs. non-preemptive), (2) the protection domain (*e.g.,* user-mode vs. kernel-mode) in which tasks within a protocol stack execute, and (3) the number of available CPUs. In general, a context switch is triggered when (1) one or more processes must sleep awaiting certain resources (such as memory buffers or I/O devices) to be come available, (2) preemptive scheduling is used and a higher priority process becomes runnable, or (3) when a currently executing process exceeds its time slice. Depending on the underlying OS and hardware platform, a context switch may be relatively time consuming due to the flushing of register windows, instruction and data caches, instruction pipelines, and translation look-aside buffers [44].
- *Synchronization Overhead* – Implementing communication protocols that execute correctly on multi-processor platforms requires synchronization mechanisms that serialize access to shared objects such as messages, message queues, protocol context records, and demultiplexing tables. Certain protocol and process architecture combinations (such as implementing connection-oriented protocols via Message Parallelism) may incur significant synchronization overhead from managing locks associated with these shared objects [42]. In addition to reducing overall throughput, synchronization bottlenecks resulting from lock contention lead to unpredictable response times that complicate the delivery of constrained-latency applications. Other sources of synchronization overhead involve contention for shared

hardware resources such as I/O buses and global memory [43]. In general, hardware contention represents an upper limit on the benefits that may accrue from multiprocessing [31].

- *Communication Overhead* – Task-based process architectures generally require some form of interprocess communication to exchange messages between protocol processing components executing on separate CPUs. Communication costs are incurred by memory-to-memory copying, message manipulation operations (such as checksum calculations and compression), and general message passing overhead resulting from synchronization and process management operations. Common techniques for minimizing communication overhead involve (1) buffer management schemes that minimize data copying [45] and attempt to preserve cache affinity properties when exchanging messages between CPUs with separate instruction and data caches, (2) integrated layer processing techniques [11], and (3) single-copy network/host interface adapters [46].
- *Load Balancing* – Certain process architectures (such as Message Parallelism) have the potential for utilizing multiple CPUs equitably, whereas others (such as Connectional, Layer, and Functional Parallelism) may under- or over-utilize the available CPUs under certain circumstances (such as bursty network and application traffic patterns or improper stage balancing).

3.1.2 The Virtual Memory (VM) Remapping Dimension

Regardless of the process architecture, minimizing the amount of memory-to-memory copying in a transport system is essential to achieve high performance [47]. In general, data copying costs provide an upper bound on application throughput [11]. As described in Section 3.2.1 below, selecting an efficient message management mechanism is one method for reducing data copying overhead. A related approach described in this section uses virtual memory optimizations to avoid copying data altogether. For example, in situations where data must be transferred from one address space to another, the kernel architecture may remap the virtual memory pages by marking their page table entries as being "copy-on-write." Copy-on-write schemes physically copy memory only if a sender or receiver changes a page's contents.

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

Page remapping schemes are often difficult to implement efficiently in the context of communication protocols, however. For example, most remapping schemes require the alignment of data in contiguous buffers that begin on page boundaries. These alignment constraints are complicated by protocol operations that significantly enlarge or shrink the size of messages. This operations include message de-encapsulation (*i.e.,* stripping headers and trailers as messages ascend through a protocol graph), presentation layer expansion [11] (*e.g.,* uncompressing or decrypting an incoming message), and variable-size header options (such as those proposed to handle TCP window scaling for long-delay paths [18]). Moreover, remapping may not be useful if the sender or receiver writes on the page immediately since a separate copy must be generated anyway [8]. In addition, for small messages, more overhead may be incurred by remapping and adjusting page table entries, compared with simply copying the data in the first place.

3.1.3 The Event Management Dimension

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

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

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

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

Alternative mechanisms for implementing event managers include *delta lists* [32], *timing wheels* [48], and heap-based [49] and list-based [8] *callout queues*. These mechanisms are built atop a hardware clock mechanism. On each "clock-tick" the event manager checks whether it is time to execute any of its registered events. If one or more events must be run, the event manager invokes the associated event handler. The different event manager mechanisms may be distinguished by the following two dimensions:

(1) Search Structure: Several search structures are commonly used to implement different event management mechanisms. One approach is to sort the events by their time-to-execute value and store them in an array. A variant on this approach (used by *delta lists* and list-based *callout queues*) replaces the array with a sorted linked list to reduce the overhead of adding or deleting an event [32]. Another approach is to use a heap-based priority queue [49] instead of a sorted list or array. In this case, the average- and worst-case time complexity for inserting or deleting an entry is reduced from $O(n)$ to $O(\lg n)$. In addition to improving average-case performance, heaps also reduce the variance of event manager operations.

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

3.2 Protocol Family Architecture Dimensions

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

3.2.1 The Message Management Dimension

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

(1) Dynamic Memory Management: Traditional data network traffic exhibits a bi-modal distribution of sizes, ranging from large messages for bulk data transfer to small messages for remote terminal access [50]. Therefore, message managers must be capable of dynamically allocating, deallocating, and coalescing fixed-sized and variable-sized blocks of memory efficiently. However, message management schemes are often tuned for a particular range of message sizes. For instance, the BSD UNIX message management facility divides its buffers into 112 byte and 1,024 byte blocks. This leads to non-uniform performance behavior when incoming and outgoing messages vary in size between small and large blocks. As discussed in [2], more uniform performance is possible if message managers support a wide range of message sizes as efficiently as they support large and/or small messages.

(2) Memory-to-memory Copy Avoidance: As mentioned in Section 3.1.2, memory-to-memory copying is a significant source of transport system overhead. Naive message managers that physically copy messages between each protocol layer are prohibitively expensive. Therefore, more sophisticated implementations avoid or minimize memory-to-memory copying via techniques such as *buffer-cut-through* [51, 52] and *lazy-evaluation* [45]. Buffer-cut-through passes messages "by reference" through multiple protocol layers to reduce copying. Likewise, lazy-evaluation techniques use reference counting and buffer-sharing to minimize unnecessary copying. These schemes may be combined with the virtual memory remapping optimizations described in Section 3.1.2.

Message managers use different methods to reduce data copying and facilitate buffer sharing. For instance, BSD and System V UNIX attach multiple buffers together to form linked-lists of message segments. Adding data to the front or rear of a buffer list does not require any data copying since it only relinks pointers. An alternative approach uses a *directed-acyclic-graph* (DAG)-based data structure [45]. A DAG allows multiple "parents" to share all or part of a message stored in a single "child." Therefore, this method improves data sharing *between* layers in a highly-layered protocol graph. This is important for reliable protocols (such as RPC or TCP) that maintain "logical" copies of messages at certain protocol layers in case retransmission is necessary.

3.2.2 The Multiplexing and Demultiplexing Dimension

Multiplexing (muxing) and demultiplexing (demuxing) select which of the sessions in an adjacent protocol layer will receive an incoming or outgoing message. A sender typically performs multiplexing, which directs outgoing messages emanating from some number of higher-layer sessions onto a smaller number of lower-layer sessions [12]. Conversely, a receiver performs demultiplexing, which directs incoming messages up to their associated sessions. Multiplexing and demultiplexing are orthogonal to data copying; depending on the message management scheme, messages need not be copied as they are multiplexed and demultiplexed throughout a protocol graph [45].

Since senders generally possess knowledge of their entire transfer context (such as message destination address(es) like connection identifiers, port numbers, and/or Internet IP addresses [11], as well as which network interfaces to use) multiplexing may be less costly than demultiplexing. In contrast, when a network adapter receives an incoming message it generally has no prior knowledge of the message's validity or eventual destination. To obtain this information, a receiver must inspect the message header and perform demultiplexing operations that select which higher-layer protocol session(s) should receive the message.

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

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

(1) Synchronization: Multiplexing and demultiplexing may occur either synchronously or asynchronously, depending primarily on whether the transport system uses a task-based or message-based process architecture. For example, message-based process architectures (such as the *x*-kernel) typically use synchronous multiplexing and demultiplexing since messages do not pass between separate process address spaces. Therefore, *intra-process* upcalls and subroutine calls are used to transfer messages up and down a protocol graph rather than more expensive asynchronous *inter-process* communication techniques such as message queues. In contrast, task-based process architectures (such as F-CSS [53]) utilize asynchronous multiplexing and demultiplexing. In this scheme, message queues are used to buffer data passed between processes that implement a layered protocol graph. Since message queues do not necessarily block the sender, it is possible to concurrently process messages in each protocol layer, which potentially increases throughput. However, this advantage may be offset by the additional context switching and data movement overhead incurred to move messages between separate CPUs [54].

(2) Layering: As shown in Figure 5 (1), multiplexing and demultiplexing may occur multiple times as messages traverse up or down a protocol graph. This *layered* approach differs from the *de-layered* approach shown in Figure 5 (2). In the de-layered approach, multiplexing and/or demultiplexing is performed only once, usually at either the highest- or lowest-layer of a protocol graph.

The use of layered multiplexing and demultiplexing provides several benefits [12]. First, it promotes modularity, since the interconnected layer components interoperate only

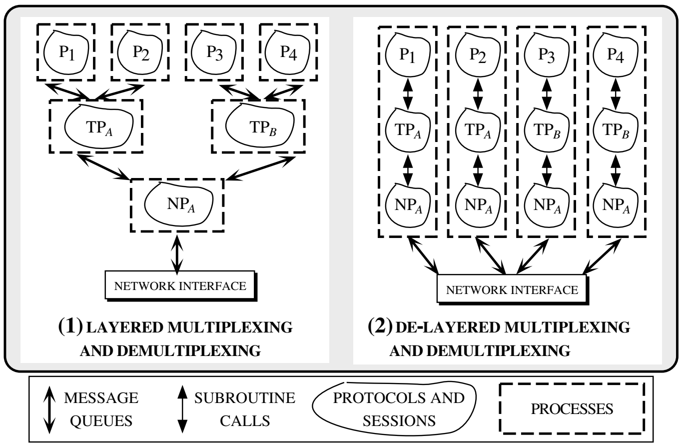

Figure 5: Layered and De-Layered Multiplexing and Demultiplexing

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

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

De-layered multiplexing and demultiplexing generally decreases jitter since there is less contention for transport system resources at a single lower-layer SAP from multiple higher-layer data streams [12]. However, the amount of context information stored within every intermediate protocol layer increases since sessions are not shared [12]. In addition, delayering expands the degree of demultiplexing at the lowest layer. This violates protocol layering characteristics found in conventional communication models (such as the ISO OSI reference model) since the lowest layer is now responsible for demultiplexing on addresses (such as connection identifiers or port numbers) that are actually associated with protocols several layers above in a protocol graph. Packet filters [39] are a technique used to address this issue. Packet filters allow applications and higher-level protocols to "program" a network interface so that particular types of incoming PDUs are demultiplexed directly to them, rather than passing through a series of intervening protocol layers first.

Note that the use of de-layered multiplexing and demultiplexing interacts with the choice of process architecture. For example, Connectional Parallelism is enhanced by protocols that demultiplex early in their protocol stack since it is difficult to maintain a strict process-per-connection association across demultiplexing boundaries [26].

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

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

(4) Caching: Several additional optimizations may be used to augment the search algorithms discussed above. These optimizations include (1) single- or multiple-item caches and (2) list reorganization heuristics that move recently accessed control blocks to the front of the search list or hash bucket-chain. A single-item cache is relatively efficient if the arrival and departure of application data exhibit "message-train" behavior. A message-train is a sequence of back-to-back messages that are all destined for the same higher-level session. However, single-item caching is insufficient if application traffic behavior is less uniform [55]. When calculating how well a particular caching scheme affects the cost of demultiplexing it is important to consider (1) the *miss ratio*, which represents how many times the desired external identifier is *not* in the cache and (2) the number of list entries that must be examined when a cache miss occurs. In general, the longer the search list, the higher the cost of a cache miss.

The choice of search algorithm and caching optimization impacts overall transport system and protocol performance significantly. When combined with caching, hashing produces a measurable improvement for searching large lists of control blocks that correspond to active network connections [2].

3.2.3 The Layer-to-Layer Flow Control Dimension

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

Layer-to-layer flow control has a significant impact on protocol performance. For instance, empirical studies [1] demonstrate the importance of matching buffer sizes and flow control strategies at each layer in the protocol family architecture. Inefficiencies may result if buffer sizes are not matched appropriately in adjacent layers, thereby causing excessive segmentation/reassembly and additional transmission delays.

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

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

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

4 Survey of Existing OS Transport System Architectures

A number of framework have emerged to simplify the development and configuration of transport systems by interconnecting session and protocol family architecture components. In general, these frameworks encourage the development of standard communication-related components (such as message managers, timer-based event dispatchers, demultiplexors [45], and assorted protocol functions [13]) by decoupling protocol processing functionality from the surrounding framework infrastructure. This section surveys the transport system architectures for the System V UNIX, BSD UNIX, *x*-kernel, and Choices operating systems. Unless otherwise noted, the systems described include System V Release 4, BSD 4.3 Tahoe, *x*-kernel 3.2, and Choices 6.16.91. Section 4.1 gives a brief summary of each system. Section 4.2 compares and contrasts each system using the taxonomy dimensions listed in Table 1.

4.1 System Overviews

This section outlines the primary software components and process architectures for each surveyed transport system in order to highlight the design decisions made by actual systems. In addition, a transport system *profile* corresponding to the taxonomy depicted in Table 1 is presented along with each overview (note that ND stands for "not defined").

4.1.1 System V STREAMS

The System V STREAMS architecture emphasizes modular components that possess uniform interfaces. It was initially developed for terminal drivers and was later extended to support network protocols and local IPC via *multiplexor drivers* and STREAM *pipes*, respectively [56]. The Table 2 illustrates the transport system profile for System V STREAMS. In the discussion below, the uppercase term "STREAMS" refers to the overall System V transport system mechanism, whereas the term "Stream" refers to a full-duplex protocol processing and data transfer path between a user application and a device driver.

As shown in Figure 6, the main components in the System V STREAMS architecture include STREAM *heads*, STREAM *modules*, STREAM *multiplexors*, and STREAM *drivers*. A STREAM head segments the user data into discrete messages. These messages are passed "downstream" from the STREAM head though zero or more STREAM modules and multiplexors to the STREAM driver, where they are transmitted by a network adapter to the appropriate network. Likewise, the driver also receives incoming messages from the network. These messages are passed "upstream" through the modules to the STREAM head, where a user process may retrieve them. STREAM modules and multiplexors may be inserted and/or removed dynamically between the head and the driver. Each module or multiplexor implements protocol processing mechanisms like encryption, compression, reliable message delivery, and routing. The following paragraphs describe each STREAMS component:

• **STREAM Heads:** STREAM heads are situated on "top" of a Stream, directly "below" the user process (as shown in Figure 6). STREAM heads provide a queueing point for exchanging data and control information between an application (running as a user process) and a Stream (running in the kernel). Each STREAM component is linked together with its adjacent components via a pair of queues: one for reading and the other for writing. These queues hold lists of

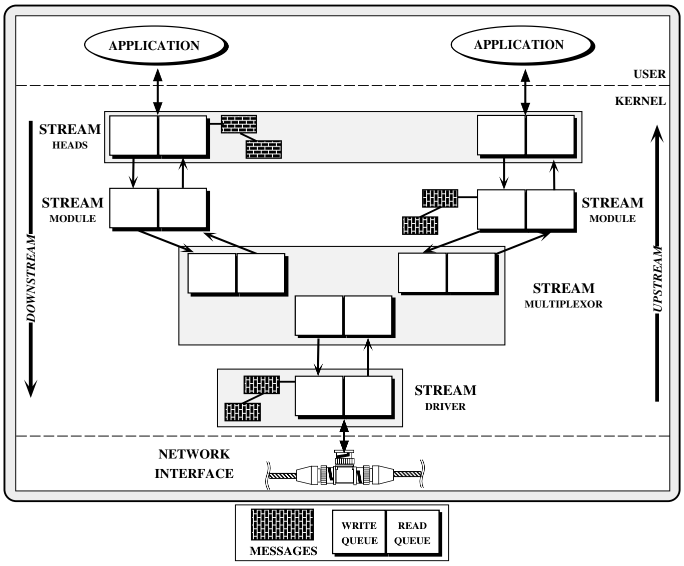

Figure 6: System V STREAMS Architecture

messages sorted by up to 256 different priority levels. Since the System V application interface does not use virtual memory remapping techniques, the STREAM head also performs memory-to-memory copying to transfer data between a user process and the kernel.

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

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

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

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

Table 2: STREAMS Profile

will have been passed to the appropriate adjacent STREAM module in the Stream. Next, the `service` routine for any STREAM modules that now have new messages in their queue(s) is scheduled to run.

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

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

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

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

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

An increasing number of STREAMS implementations [28, 38, 25, 30] utilize shared memory, symmetric multiprocessing capabilities within a multi-threaded kernel address space. These implementations supports various levels of STREAMS concurrency. These concurrency levels range from relatively fine-grain parallelism (such as *queue-level* with one light-weight process (LWP) for the STREAM module read queue and one LWP for the STREAM module write queue and *queue-pair-level* with one LWP shared by a STREAM module queue pair) to more coarse-grained approaches (such as *module-level* with one LWP shared across all instances of a STREAM module and *module-class-level* with one LWP shared across a particular class of STREAM modules).

4.1.2 BSD UNIX Network Subsystem

BSD UNIX provides a transport system framework that supports multiple protocol families such as the Internet, XNS, and OSI protocols [8]. BSD provides a general-purpose application interface called *sockets*. Sockets allow bi-directional communication of arbitrary amounts of data between unrelated processes on local and remote hosts. Table 3 illustrates the transport system profile for BSD UNIX.

The concept of a *communication domain* is central to BSD's multiple protocol family design. A domain specifies both a protocol family and an address family. Each protocol family implements a set of protocols corresponding to standard socket types in the domain (*e.g.,* `SOCK_STREAM` for reliable byte-stream communication and `SOCK_DGRAM` for unreliable datagram communication). An address family defines an address format (*e.g.,* the address size in bytes, number and type of fields, and order of fields) and a set of kernel-resident subroutines that interpret the address format (*e.g.,* to determine which subnet an IP message is intended for). The standard BSD release supports address families for the Internet domain, XEROX NS domain, OSI domain, and UNIX domain (which only exchanges information between sockets in processes on a local host).

There are three main layers in the BSD transport system design: the *socket layer*, *protocol layer*, and *network in-*

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

Table 3: BSD UNIX Profile

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

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

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

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

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

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

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

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

4.1.3 x-kernel

The *x*-kernel is a modular, extensible transport system kernel architecture designed to support prototyping and experimentation with alternative protocol and session architectures [2]. It was developed to demonstrate that layering and modularity are not inherently detrimental to network protocol performance [2]. The *x*-kernel supports protocol graphs that implement a wide range of standard and experimental protocol families, including TCP/IP, Sun RPC, Sprite RCP, VMTP, NFS, and Psync [57]. Unlike BSD UNIX, whose protocol family architecture is characterized by a static, relatively monolithic protocol graph, the *x*-kernel supports dynamic, highly-layered protocol graphs. Table 4 illustrates the transport system profile for the *x*-kernel.

The *x*-kernel's protocol family architecture provides highly uniform interfaces to its mechanisms, which manage three communication abstractions that comprise protocol graphs [2]: *protocol objects*, *session objects*, and *message objects*. These abstractions are supported by other reusable software components that include a *message manager* (an abstract data type that encapsulates messages exchanged between session and protocol objects), a *map manager* (used for demultiplexing incoming messages between adjacent protocols and sessions), and an *event manager* (based upon *timing wheels* [48] and used for timer-driven activities like TCP's adaptive retransmission algorithm). In addition, the *x*-kernel provides a standard library of *micro-protocols*. These are reusable, modular software components that implement mechanisms common to many protocols (such as include sliding window transmission and adaptive retransmission schemes, request-response RPC mechanisms, and a "blast" protocol that uses selective retransmission to reduce channel utilization [10]). The following paragraphs describe the *x*-kernel's primary software components:

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

• **Session Objects:** A session object maintains context information associated with a local end-point of a connection. For example, a session object stores the context information for an active TCP state machine. Protocol objects create and destroy session objects dynamically. When an application opens multiple connections, one or more session objects will be created within the appropriate protocol objects in a protocol graph. The *x*-kernel supports operations on session objects that involve "layer-to-layer" activities such as exchanging messages between higher-level and lower-level sessions. However, the *x*-kernel's protocol family architecture framework does not provide standard mechanisms for "end-to-end" session architecture activities such as connection management, error detection, or end-to-end flow control. A related project, Avoca, builds upon the basic *x*-kernel facilities to provide these end-to-end session services [10].

• **Message Objects:** Message objects encapsulate control and user data information that flows "upwards" or "downwards" through a graph of session and protocol objects. In order to decrease memory-to-memory copying and to implement message operations efficiently, message objects are implemented using a "directed-acyclic-graph" (DAG)-based data structure. This DAG-based scheme uses "lazy-evaluation" to avoid unnecessary data copying when passing messages between protocol layers [45]. It also stores message headers in a separate "header stack" and uses pointer arithmetic on this stack to reduce the cost of prepending or stripping message headers.

The *x*-kernel employs a "process-per-message" message-based process architecture that resides in either the OS kernel or in user-space. The kernel implementation maintains a pool of light-weight processes (LWPs). When a message arrives at a network interface, a separate LWP is dispatched from the pool to shepard the message upwards through the graph of protocol and session objects. In general, only one context switch is required to shepard a message through the protocol graph, regardless of the number of intervening protocol layers. The *x*-kernel also supports other context switch optimizations that (1) allow user processes to transform into kernel processes via system calls when sending message and (2) allow kernel processes to transform into user processes via upcalls when receiving messages [58].

4.1.4 The Conduit Framework

The Conduit provides the protocol family architecture, session architecture, and application interface for the Choices operating system [59]. Choices was developed to investigate the suitability of object-oriented techniques for designing

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

Table 4: *x*-kernel Profile

and implementing OS kernel and networking mechanisms.[5] For example, the design of ZOOT (the Choices TCP/IP implementation) uses object-oriented language constructs and design methods such as inheritance, dynamic binding, and delegation [60] to implement the TCP state machine in a highly modular fashion. Together, Choices and the Conduit provide a general-purpose transport system. Table 5 illustrates the transport system profile for the Choices Conduit. In the discussion below, the term "Conduit" refers to the overall transport system, whereas a "`Conduit`" corresponds to an abstract data type used to construct and coordinate various network protocols.

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

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

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

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

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

The Conduit is implemented in user-space and the relationship of processes to `Conduits` and `Conduit Messages` is a hybrid between message-based and task-based process architectures. Messages are escorted through the Conduit

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

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

Table 5: Conduit Profile

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

4.2 Transport System Comparisons

This section compares and contrasts the four surveyed transport systems using the taxonomy dimensions and alternatives presented in Table 1. Section 4.2.1 focuses on the kernel architecture dimensions described in Section 3.1 and Section 4.2.2 focuses on the protocol family architecture dimensions described in Section 3.2.

4.2.1 Comparison of Kernel Architecture Dimensions

The Process Architecture Dimension: The surveyed transport systems exhibit a range of process architectures. The conventional System V STREAMS implementation uses a variant of the task-based process architecture known as a "process-per-module" approach. However, as described in Section 4.1.1, the standard System V STREAMS approach does not associate a heavy-weight OS process per module in an effort to reduce context switching overhead and minimize kernel state information required for process management.

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

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

The Virtual Memory Remapping Dimension: Recent versions of *x*-kernel provide virtual memory remapping [45] for transferring messages between application process and the kernel. The Conduit, System V STREAMS and BSD UNIX, on the other hand, do not generally provide this support.

The Event Management Dimension: BSD UNIX stores pointers to subroutines in a *linked-list callout queue*. These preregistered subroutines are called when a timer expires. System V, on the other hand, maintains a *heap-based callout table*, rather than a sorted list or array. The heap-based implementation outperforms the linked-list approach under heavy loads [49]. The *x*-kernel uses *timing wheels* [48] instead of callout lists or heaps.

4.2.2 Comparison of Protocol Family Architecture Dimensions

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

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

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

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

The BSD UNIX multiplexing and demultiplexing mechanisms differ depending on which protocol component and protocol family are involved. For instance, its IP implementation uses the 8-bit IP message type-of-service field to index into an array containing 256 entries that correspond to higher-layer protocol control structures. On the other hand, its TCP implementation uses sequential search with a one-item cache to demultiplex incoming messages to the appropriate connection session. As described in Section 3.2.2, this implementation is inefficient when application data arrival patterns do not form message-trains [55].

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

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

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

5 Summary

This paper examines the major levels of abstraction in the transport system architecture. A taxonomy of six key transport system mechanisms is presented and used to compare different design alternatives found in four existing commercial and experimental operating systems. Our research group at University of California, Irvine is currently using this taxonomy to guide the development of a highly modular transport system development environment called called ADAPTIVE [61].

ADAPTIVE is an integrated collection of communication-related C++ components [62] that may be combined via inheritance, template instantiation, and object composition. These components help control for factors (such as concurrency control schemes, protocol functionality, and application traffic characteristics) that significantly affect transport system performance in a shared memory, symmetric multi-processor environment [54]. We are building ADAPTIVE to facilitate experimentation with various strategies for developing transport systems that operate efficiently across high-performance networks.

Based upon our experience with transport systems, combined with our survey of research literature and existing systems, we view the following as important open research issues pertinent to the development of transport system architectures:

- Which transport system levels (*e.g.,* application interface, session architecture, protocol family architecture, kernel architecture) incur the most communication performance overhead?
- Which choices from among the taxonomy dimensions and alternatives improve the overall communication performance? For example, which process architectures result in the highest performance? Likewise, what combinations of application requirements and network characteristics are most suitable for different transport system profiles?

- How will the performance bottlenecks shift as the boundary between hardware and software changes? For instance, the high cost of message management operations such as fragmentation and reassembly may be greatly reduced if they are performed in hardware, as proposed for ATM.
- Which transport system profiles are best suited for multimedia applications running in high-performance network environments? Moreover, what are the appropriate design strategies and implementation techniques required to provide *integrated* support for multimedia applications that run on general-purpose workstation operating systems?

Much additional empirical research is necessary to address these research questions adequately. We hope this paper helps to clarify essential issues and relationships that arise when designing high-performance transport system architectures.

References

[1] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica, "Is Layering Harmful?," *IEEE Network Magazine*, January 1992.

[2] 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.

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

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

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

[6] J. Sterbenz and G. Parulkar, "AXON: Application-Oriented Lightweight Transport Protocol Design," in *International Conference on Computers and Communications*, (New Delhi, India), Nov. 1990.

[7] UNIX Software Operations, *UNIX System V Release 4 Programmer's Guide: STREAMS*. Prentice Hall, 1990.

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

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

[10] S. W. O'Malley and L. L. Peterson, "A Dynamic Network Architecture," *ACM Transactions on Computer Systems*, vol. 10, pp. 110–143, May 1992.

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

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

[13] 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, Sept. 1993.

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

[15] M. S. Atkins, S. T. Chanson, and J. B. Robinson, "LNTP – An Efficient Transport Protocol for Local Area Networks," in *Proceedings of the Conference on Global Communications (GLOBECOM)*, pp. 705–710, 1988.

[16] D. R. Cheriton, "UIO: A Uniform I/O System Interface for Distributed Systems," *ACM Transactions on Computer Systems*, vol. 5, pp. 12–46, Feb. 1987.

[17] M. D. Maggio and D. W. Krumme, "A Flexible System Call Interface for Interprocess Communication in a Distributed Memory Multicomputer," *Operating Systems Review*, vol. 25, pp. 4–21, April 1991.

[18] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance," *Network Information Center RFC 1323*, pp. 1–37, Oct. 1992.

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

[20] W. Doeringer, D. Dykeman, M. Kaiserswerth, B. Meister, H. Rudin, and R. Williamson, "A Survey of Light-Weight Transport Protocols for High-Speed Networks," *IEEE Transactions on Communication*, vol. 38, pp. 2025–2039, November 1990.

[21] L. Svobodova, "Implementing OSI Systems," *IEEE Journal on Selected Areas in Communications*, vol. SAC-7, pp. 1115–1130, Sept. 1989.

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

[23] E. C. Cooper, P. A. Steenkiste, R. D. Sansom, and B. D. Zill, "Protocol Implementation on the Nectar Communication Processor," in *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, (Philadelphia, PA), pp. 135–144, ACM, Sept. 1990.

[24] 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.

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

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

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

[28] 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.

[29] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activiation: Effective Kernel Support for the User-Level Management of Parallelism," *ACM Transactions on Computer Systems*, pp. 53–79, February 1992.

[30] 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.

[31] 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.

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

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

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

[35] 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.

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

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

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

[39] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 259–270, Jan. 1993.

[40] 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.

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

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

[43] 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.

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

[45] 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, September 1989.

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

[47] 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.

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

[49] R. E. Barkley and T. P. Lee, "A Heap-Based Callout Implementation to Meet Real-Time Needs," in *Proceedings of the USENIX Summer Conference*, pp. 213–222, USENIX Association, June 1988.

[50] R. Caceres, P. Danzig, S. Jamin, and D. Mitzel, "Characteristics of Wide-Area TCP/IP Conversations," in *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, (Zurich Switzerland), pp. 101–112, ACM, Sept. 1991.

[51] 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.

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

[53] 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.

[54] D. C. Schmidt, "ASX: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[55] P. E. McKenney and K. F. Dove, "Efficient Demultiplexing of Incoming TCP Packets," Tech. Rep. SQN TR92-01, Sequent Computer Systems, Inc., December 1991.

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

[57] L. L. Peterson, N. Buchholz, and R. D. Schlichting, "Preserving and Using Context Information in Interprocess Communication," *ACM Transactions on Computer Systems*, vol. 7, pp. 217–246, August 1989.

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

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

[60] J. M. Zweig and R. Johnson, "Delegation in C++," *Journal of Object-Oriented Programming*, vol. 4, pp. 31–34, November/December 1991.

[61] 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.

[62] D. C. Schmidt and T. Suda, "The ADAPTIVE Service eXecutive: an Object-Oriented Architecture for Configuring Concurrent Distributed Applications," in *Proceedings of the* 8^{th} *International Working Conference on Upper Layer Protocols, Architectures, and Applications*, (Barcelona, Spain), IFIP, June 1994.