Computing Infrastructure Issues in Distributed Communications Systems

A Survey of Operating System Transport System Architectures

Technical Report 92-26

Douglas C. Schmidt and Tatsuya Suda

schmidt@ics.uci.edu and suda@ics.uci.edu
Department of Information and Computer Science,
University of California, Irvine,
Irvine, CA 92717, U.S.A.
(714) 856-4105 (phone)
(714) 856-4056 (fax) [1]

[1]This material is based upon work supported by the National Science Foundation under Grant No. NCR-8907909. This research is also supported in part by the University of California MICRO program.

Contents

List of Figures

List of Tables

Abstract

The performance of distributed applications (such as file transfer, remote login, tele-conferencing, full-motion video, and scientific visualization) is influenced by several factors that interact in complex ways. In particular, application performance is significantly affected both by communication infrastructure factors and computing infrastructure factors. Several communication infrastructure factors include channel speed, bit-error rate, and congestion at intermediate switching nodes. Computing infrastructure factors include (among other things) both protocol processing activities (such as connection management, flow control, error detection, and retransmission) and general operating system factors (such as memory latency, CPU speed, interrupt and context switching overhead, process architecture, and message buffering). Due to a several orders of magnitude increase in network channel speed and an increase in application diversity, performance bottlenecks are shifting from the network factors to the transport system factors.

This paper defines an abstraction called an "Operating System Transport System Architecture" (OSTSA) that is used to classify the major components and services in the computing infrastructure. End-to-end network protocols such as TCP, TP4, VMTP, XTP, and Delta-t typically run on general-purpose computers, where they utilize various operating system resources such as processors, virtual memory, and network controllers. The OSTSA provides services that integrate these resources to support distributed applications running on local and wide area networks.

A taxonomy is presented to evaluate OSTSAs in terms of their support for protocol processing activities. We use this taxonomy to compare and contrast five general-purpose commercial and experimental operating systems including System V UNIX, BSD UNIX, the *x*-kernel, Choices, and Xinu.

1 Introduction

In the past few years, the demand for many kinds of communication services has intensified. Distributed applications involving voice, video, data, and images are rapidly expanding, and application requirements and usage patterns are undergoing significant qualitative and quantitative changes. For instance, multimedia applications such as medical imaging, supercomputer graphics for scientific visualization, and tele-conferencing have communication requirements that differ greatly from traditional data applications like remote login, email, and file transfer [Che86].

Qualitative changes in application requirements necessitate extremely high throughput (*e.g.*, HDTV), strict real-time delivery (*e.g.*, robotics), low delay and low delay jitter (*e.g.*, voice conversation), multicast capability (*e.g.*, video-conferencing), and some degree of loss tolerance (*e.g.*, hierarchically coded voice and video). In addition, distributed applications impose different network traffic patterns. For example, some applications generate highly bursty traffic (*e.g.*, variable bit-rate video applications), some generate continuous traffic (*e.g.*, constant bit-rate video applications), and others generate short, interactive, transaction-oriented traffic (*e.g.*, network file systems using remote procedure calls (RPC)).

Quantitative changes in distributed computing usage are also occurring. For instance, in current workstation environments, local computing activities (*e.g.*, editing and compiling) dominate remote communications (which consist mostly of network file system operations). In future multimedia workstation environments, on the other hand, it is expected that remote communication activities (*e.g.*, audio- and video-conferencing applications) will dominate local computing.

Many researchers, commercial vendors, and standards bodies are working to integrate lightwave communication technology with general-purpose computer systems. For example, in very high speed internet (VHSI) [Par90] environments, these distributed systems will consist of high-speed public access networks linked to high-speed LANs and MANs [Gre91]. Support for multimedia applications running on these distributed systems is provided by both the *communication infrastructure* and the *computing infrastructure* (see Figure 1).

The communication infrastructure provides mechanisms (*e.g.*, transmission media and the lower three layers of the ISO OSI network protocols) for transmitting information throughout a network. The computing infrastructure is more precisely defined as the "OSTSA" (Operating System Transport System Architecture[1]) in this paper. It integrates peer-to-peer network protocols into general-purpose computer host operating systems (containing OS kernel services and hardware devices) to support diverse user applications running across the communication infrastructures.

The communication infrastructure now exhibits very high transfer rates due to recent advances in optical transmission technology. Example communication infrastructures include the Fiber Distributed Data Interface (FDDI), the Distributed Queue Dual Bus (DQDB), and the Asynchronous Transfer Mode (ATM). These new technologies, coupled with an increase in application diversity) have shifted performance bottlenecks to the OSTSA computing infrastructure [CJRS89].

The OSTSA computing infrastructure consists of components that operate at several levels of abstraction. First, it provides user processes with an interface to end-to-end network protocols such as TCP, TP4, and/or XTP. These protocols implement various transport service classes that support communication between distributed user applications. Next, it provides a framework[2] for orchestrating various resources managed by the OS to support these network protocols. These resources include hardware devices (such as CPU, primary and secondary storage, and high-speed I/O devices like network controllers) and software abstractions (such as virtual memory, processes, and protocol

[1] An architecture is a design that describes the system components, provides a functional decomposition, and specifies individual module semantics.

[2] A *framework* is defined as "a design that can be reused" [Zwe91].

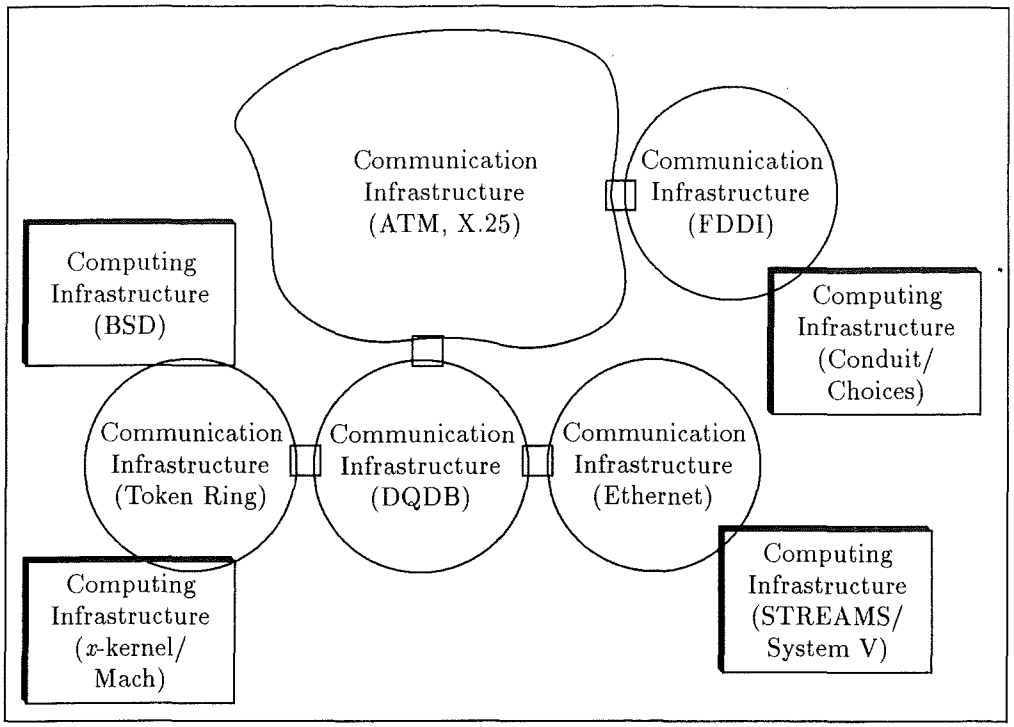

Figure 1: Examples of Computing and Communications Infrastructures

graphs[3]).

Next generation OSTSAs must be flexible in order to meet diverse application requirements and to take advantage of advances in the communication infrastructure. However, existing OSTSAs are the performance bottleneck in high-speed networks with channel speeds exceeding 100 Mbps [CJRS89]. This bottleneck is manifested by the *throughput preservation problem* [MS92], where only a limited fraction of the available network bandwidth is delivered to distributed applications. This situation results from OSTSA overhead (such as memory-to-memory copying and process management operations like interrupt handling, context switching, and scheduling) not decreasing as rapidly as the transmission media channel-speed is increasing. Moreover, the throughput preservation problem persists despite an increase in CPU speeds.[4]

This paper presents a taxonomy of the major OSTSA dimensions and compares and contrasts five general-purpose commercial and experimental OSTSAs (including System V UNIX, BSD UNIX, *x*-kernel, Choices, and Xinu)[5] along the taxonomy dimensions. We focus on general-purpose OSTSAs in this paper for several reasons. First, they facilitate flexibility and extensibility and thereby

[3] A protocol graph is a generalization of a protocol stack; it represents the hierarchical relations between protocols in one or more protocol suites [OP91]. Figure 3 in Section 2 depicts an example protocol graph containing certain Internet and OSI protocols.

[4] There are several explanations for this: (1) networks have increased by 5 or 6 orders of magnitude (from kbps to Gbps), whereas CPU speeds have only increased by 2 or 3 orders of magnitude (from 1 MIP up to 100 MIPS) [Haa91], (2) network host interfaces in existing systems interrupt the CPU for every packet transmitted [Haa91], and (3) despite leading to an increase in total MIPS, RISC architectures (such as the SPARC) penalize this interrupt-driven network communications, since they typically have higher context switching overhead, resulting from the cost of flushing instruction and data caches and pipelines, storing and retrieving large register windows, etc. [Ste92].

[5] This paper describes System V Release 4, 4.3 BSD, *x*-kernel 3.2, Choices 6.16.91, and Xinu version 7 unless otherwise noted.

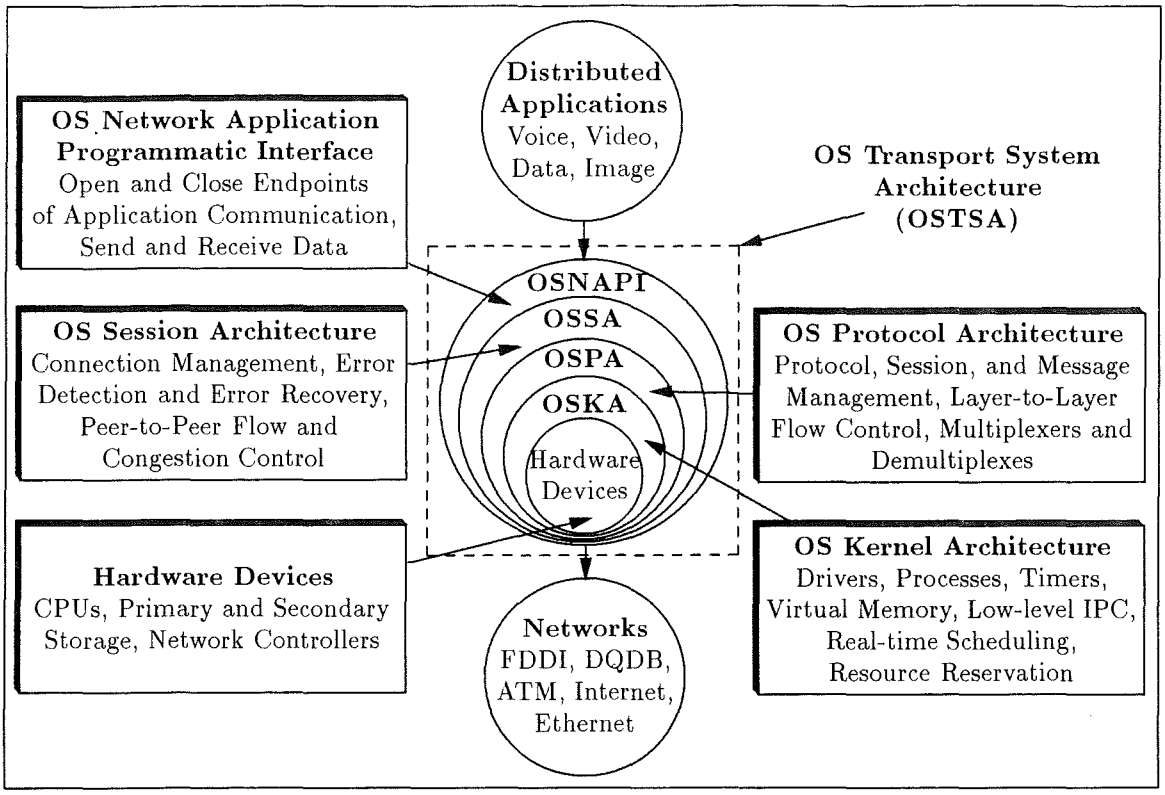

Figure 2: The OS Transport System Architecture

encourage experimentation.[6] Second, special-purpose solutions (such as off-board processors [KC88] and VLSI-based hardware implementations [Che89]) may not be adaptive enough to meet multimedia application requirement diversity. For instance, a special-purpose solution that efficiently supports one application or type of network is not necessarily appropriate for other applications coexisting on the same network. Third, even if special multi-processor pool architectures [JSB90] or off-board processors become widely available, they must still interoperate with the host operating system at some point. Studies [KC88] have shown that significant host OS and protocol processing overhead remains, despite using off-board protocol processors like the VMP Network Adapter Board. Therefore the OSTSA dimensions described in this paper remain an integral part of the overall throughput preservation problem.

The remainder of the paper is organized as follows: Section 2 describes the OSTSA components in detail; Section 3 presents a general taxonomy for classifying OSTSAs; Section 4 provides an in-depth survey of five representative OSTSAs; Section 5 summarizes the paper and outlines several important open research issues.

2 OS Transport System Architecture Components

Operating System Transport System Architectures (OSTSAs) provide a framework that coordinates various hardware resources (*e.g.*, primary and secondary storage and CPU(s)) and software abstrac-

[6] Experimentation is important since there is no clear consensus on precisely how different factors affect OSTSA performance [PC91]. Controlled empirical experimentation, based on a general-purpose OSTSA, is a useful method for investigating the impact of various performance factors.

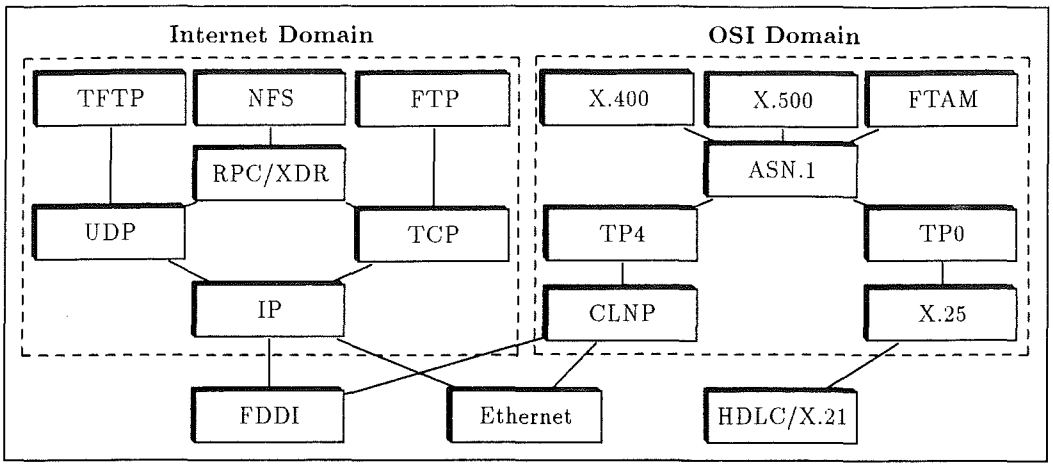

Figure 3: Typical Protocol Graph for Internet and OSI Protocol Families

tions (*e.g.*, algorithms and data structures that represent process architectures and protocol graphs) to support network protocols.

OSTSAs are frequently modeled as *virtual machines*, representing the different levels of abstraction they encompass [Tan88, Tan90, Tan92]. Each virtual machine level is characterized by the service interfaces it exports to the levels surrounding it. Protocol suites like TCP/IP or OSI are implemented by combining various services offered by the nested OSTSA levels shown in Figure 2.[7] The following paragraphs provide a brief overview of all the OSTSA levels discussed in this paper.

The outermost-level of the OSTSA is the Operating System Network Application Programmatic Interface (OSNAPI). The OSNAPI provides service interfaces through which user processes (*e.g.*, distributed multimedia applications) interact with inner-level OSTSA services. The OSNAPI provides data-transfer operations (*e.g.*, sending and receiving messages) and control operations (*e.g.*, connection establishment and termination) to user applications. The BSD UNIX socket layer is a representative example of an OSNAPI.

The second outer-most level of the OSTSA is the Operating System Session[8] Architecture (OSSA). The OSSA provides peer-to-peer network protocol services. These services are associated with protocol *sessions* that contain information used to manage the state of end-to-end *network connections*.[9] OSSA services include dynamically establishing and terminating network connections, managing protocol interpreters (*e.g.*, the TCP or XTP finite state machine) for network connections, controlling peer-to-peer flow and congestion, and providing various error detection and error recovery policies.

The Operating System Protocol Architecture (OSPA) provides services that compose and manage multiple protocol graphs. These protocol graphs implement protocol suites such as TCP/IP or OSI (shown in Figure 3). Each layer in the protocol graph consists of one or more network protocols (*e.g.*, RPC/XDR, TCP, IP, TP4, and CLNP). These services include intra-protocol services (such

[7]The virtual machine model depicted in Figure 2 is used for descriptive purposes throughout this paper. Its levels represent an abstraction of the services and interfaces common to many existing OSTSAs, although not all systems follow such a strict hierarchy, and may bypass or omit certain levels. In particular, OSTSA implementations often proceed in a monolithic, non-uniform manner for performance reasons [CT90, Ten89] (as Section 3.3.1 discusses below).

[8]The term "session" is used throughout the paper to refer to the data structures and subroutines that implement a network connection. It is not equivalent in meaning to the ISO OSI "session layer."

[9]Note that creating a new session only involves processing on the local host (such as dynamically allocating a session control block and associating it with the appropriate protocol component). Establishing a network connection, on the other hand, usually involves a message handshake exchange with peer protocols and sessions located on remote hosts.

as session management and message management) and inter-protocol services (such as layer-to-layer flow control and multiplexing and demultiplexing). The primary distinction between the OSSA and OSPA levels is that OSSA services manage the session state information for a particular network connection, whereas OSPA services manage multi-layered protocol graphs (with each protocol layer containing one or more sessions).

The Operating System Kernel Architecture (OSKA) provides services that manage hardware resources such as primary and secondary storage, CPU(s), and various I/O devices. These services include concurrent programming abstractions and multi-processing support, timer handling, virtual memory management, and low-level interprocess communication (IPC). In addition, to support delay-sensitive user applications, the OSKA may provide mechanisms such as real-time scheduling and resource reservation [PPA+90, GA91, AH91]. The primary distinction between the OSPA and OSKA levels is that OSKA services are also utilized by user application programs and other OS subsystems such as the file system. On the other hand, OSPA services pertain primarily to network protocols and distributed applications.

At the core of the OSTSA are hardware devices such as CPU(s), memory hierarchies (*e.g.*, instruction and data caches, main memory, magnetic and optical disks, and magnetic tape), and network controllers (which are responsible for transmitting bit streams into a network). For instance, Ethernet network controllers mediate access to the logical link layer, providing services like frame transmission and reception (*e.g.*, using scatter-read and gather-write and direct memory access (DMA)), determining link layer addresses, and collision detection. Sections 2.1 through 2.4 discuss each level of the OSTSA in greater detail.

2.1 The OS Network Application Programmatic Interface (OSNAPI)

The OS Network Application Programmatic Interface (OSNAPI) provides interfaces that enable user processes (*i.e.*, processes executing in user-space) to access inner-level OSTSA services in order to exchange data and control messages with peer entities on remote hosts. Example OSNAPIs include BSD sockets [LMKQ89], the System V Transport Layer Interface (TLI) [Sun90], the V kernel's UIO system [Che87], and the Multifaceted Communications System for the NCUBE [MK91].

OSNAPIs provide service interfaces for transferring data. Since operations on network connections are very similar to operations on files and other I/O devices, OSNAPI service interfaces often supply some variant on the standard `open`, `close`, `read`, and `write` paradigm used by traditional I/O interfaces. Typical services for sending and receiving data include synchronous and/or asynchronous I/O, buffered and/or unbuffered I/O, scatter/gather I/O, blocking and/or non-blocking I/O, multi-priority in-band and/or out-of-band I/O, and multicast and/or broadcast I/O [MK91]. An important evaluation criteria for an OSNAPI is how efficiently it supports all these different types of I/O services.

Supporting control operations is another important OSNAPI service. Control operations provide user applications with interfaces for services that do not directly involve data transfer. These services include establishing and terminating connections (*e.g.*, using in-band and out-of-band signaling), dynamically configuring protocol graphs (*e.g.*, pushing a System V STREAM module onto a Stream), and setting and retrieving the values for user-tunable options (*e.g.*, the maximum datagram size and send/receive buffer sizes).

The OSNAPI level often accounts for a large portion of the overall OSTSA performance overhead. For example, [HP91] profiled BSD UNIX and determined that 31 percent of the total user-to-user latency for TCP applications resulted from socket layer processing overhead. Likewise, [JSB90] described how BSD socket processing required 36 percent of the total message processing time for outgoing messages and 43 percent for incoming messages.

This overhead results from several factors. First, memory-to-memory copying is often performed

at the OSNAPI level, in order to move messages from user-space to kernel-space and vice versa (Sections 3.1.3 and 3.2.1 discuss mechanisms for reducing this memory copying overhead). Second, distributed applications typically run in "user-mode," *i.e.*, outside the OS kernel address space. Therefore, the kernel must perform one or more context switches to transfer messages from the network device driver interfaces, up through the protocol graph to the OSNAPI-level, where user applications may be waiting to receive them [MRA87]. [Che87, MK91, HP91] evaluate the functionality and performance of alternative OSNAPIs. A thorough discussion of OSNAPI issues is beyond the scope of this paper.

2.2 The OS Session Architecture (OSSA)

The OS Session Architecture (OSSA) provides a framework for implementing peer-to-peer network protocol services that manage the state of protocol *sessions*. Sessions are used to implement network connections. Conceptually, the OSSA exists within the framework provided by the OSPA and the OSKA (described in Sections 2.3 and 2.4, respectively).

The OSSA provides services that include *connection management* (*e.g.*, opening and closing connections, and reporting and updating connection status information), managing *protocol interpreters* (*e.g.*, controlling transitions between states in the TCP finite state machine) for active network connections, controlling peer-to-peer *flow and congestion*[10] (*e.g.*, advertizing the available sliding window size, and tracking round-trip delays), *error detection and recovery* (*e.g.*, computing checksums, detecting mis-sequenced or duplicated messages, and performing acknowledgments and retransmissions), and *quality-of-service negotiation* (*e.g.*, throughput, delay, error rate, jitter and priority characteristics) with peer sessions entities.

The performance of OSSA services largely depends on the complexity and characteristics of the protocols they support. For example, [CT90] reports that the complex processing of presentation layer conversions (which involves encoding and/or decoding binary messages using the ASN.1 transfer syntax [HD89]) accounts for over 90 percent of the OSSA overhead. In addition, OSSA performance is also affected by protocol characteristics such as the *transmitted segment size* (larger segments decrease the ratio of header to data overhead and also reduce the number of subroutine calls, interrupts, and context switches to move messages between protocol layers), *peer-to-peer flow and congestion control algorithms* (*e.g.*, sliding window versus rate control), and *connection management schemes* (*e.g.*, implicit timer-based connections vs explicit handshaking).

Recent research has addressed several OSSA-related issues. Avoca [OP90a, BO91] used the *x*-kernel as a run-time environment to investigate the performance characteristics and reuse potential from using modular, highly-layered protocols and sessions. The Conduit framework [Zwe90, Zwe91] from the Choices OS is used to investigate the applicability of object-oriented programming techniques such as *delegation* [ZJ91] and *inheritance* to network protocol and session design and implementation. Finally, the ADAPTIVE system [BSS92] is being developed to identify OSSA, OSPA, and OSKA configurations that efficiently support diverse multimedia applications running on a wide range of high-speed networks. [PS91, DDK+90] survey various OSSA issues in greater detail; a thorough discussion of the issues is beyond the scope of this paper.

[10] Different flow control mechanisms are used for different OSTSA levels. For example, *peer-to-peer* flow control synchronizes the rate of senders and receivers communicating at the same protocol layer (*e.g.*, between two TCP connections residing on different hosts); *layer-to-layer* flow control regulates the amount of data exchanged between adjacent layers in a protocol graph (*e.g.*, between TCP and IP STREAM modules in System V STREAMS).

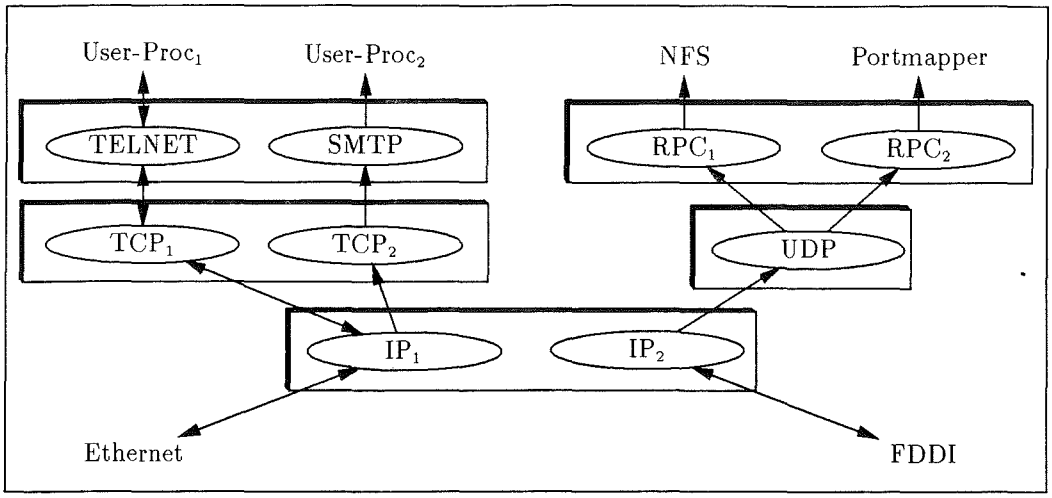

Figure 4: Example Session Graph

2.3 The OS Protocol Architecture (OSPA)

The previous section described OSSA services that manage the session state information associated with an active network connection. The OS Protocol Architecture (OSPA), on the other hand, constitutes a broader framework that supports network computing services that occur *within* and *between* the layers in a protocol graph. Services within a given protocol layer involve *creating and destroying* protocol sessions (*e.g.*, in response to a user application performing socket `accept` or `close` system calls in BSD UNIX). Services between adjacent protocol layers involve regulating *layer-to-layer data flow* (*e.g.*, the `canput` subroutine in the System V STREAMS utility library that determines if there is available space left in a message queue), along with *multiplexing/demultiplexing* and *encapsulating/de-encapsulating* outgoing and incoming messages.

Each network protocol contains one or more OSSA sessions. A session is typically created and destroyed by an OSPA-level "protocol session management" facility. Figure 4 illustrates a session graph corresponding to several protocol layers. In the figure, multiple sessions (shown in the ovals) are encapsulated by protocol components (represented by the rectangles). The OSSA manages the sessions, whereas the OSPA manages the protocols.

To facilitate interoperability, OSPAs may support multiple protocol suites simultaneously. For example, the BSD UNIX networking subsystem supports three different OSPAs that implement the TCP/IP, OSI, and XNS protocol suites [LMKQ89]. The performance of OSPA services depends heavily on how they are integrated with OS Kernel Architecture (OSKA) services [Cla82] (particularly the *process architecture* described in Section 3.1.1). OSPA factors that affect overall OSTSA performance overhead involve creating, executing, and synchronizing protocol and session components. Other potentially expensive OSPA services include message management (*e.g.*, adding and stripping headers/trailers and fragmenting/reassembling messages) and the multiplexing and demultiplexing of messages to the appropriate protocol or session.

Section 3.2 describes the OSPA level in greater detail. Section 4 surveys five OSTSAs (System V UNIX STREAMS [UNI90], the BSD protocol layers [LMKQ89], the *x*-kernel [HP91], Choices' Conduit framework [Zwe90], and the Xinu TCP/IP subsystem [Com91b]) that provide different types of OS Protocol Architectures.

2.4 The OS Kernel Architecture (OSKA)

The OSNAPI, OSSA, and OSPA levels described above ultimately interoperate with the services provided by the Operating System Kernel Architecture (OSKA). The OSKA provides services such as *virtual memory management* (*e.g.* allocating and deallocating memory objects), *concurrent programming abstractions* for uni- and multi-processors (*e.g.*, creating, scheduling, executing, synchronizing, and destroying heavy-weight or light-weight processes), sending and receiving *low-level IPC messages*[11] between communicating processes, *event management* (*e.g.*, registering, canceling, and invoking subroutines under timer-control), and *interrupt handling* (*e.g.*, servicing device interrupts from network controllers and disk controllers).

Many researchers [Cla82, HP91, WM87, CT90] suggest that of all the OSTSA levels, the OSKA services have the greatest overall impact on end-to-end protocol performance. For example, [PS91] reports that OSPA and OSSA services generally account for less than 20 percent of all protocol processing time, and that the remaining time is spent performing OSKA services like interrupt and event handling, copying data, and process management.

OSKA performance is significantly affected by *process management* services that include creating, scheduling, executing, and synchronizing multiple OSKA processes. Process management is generally time-consuming. For example, [Mul90] states that 36 percent of the overall Amoeba RPC round-trip delay is related to client and server process scheduling overhead. *Interrupt-handling* and *context switching* are two high-cost process management activities. Interrupts are used by network devices to inform the OS protocol software that incoming messages are available for higher-layer protocol processing. Interrupts often lead to a context switch, which represents a major performance penalty [Cla82]. Another source of context switching overhead occurs from invalidating virtual memory *translation-lookaside buffers* [JSB90].

Section 3.1 describes the OSKA level in greater detail. Some representative OSKAs include the Mach micro-kernel [GDFR90], the *x*-kernel (the *x*-kernel provides both OSKA and OSPA services) [HP91], the V-kernel [Che88], BSD UNIX [LMKQ89] and System V UNIX [Bac86]. [TR85, Gos91, GL89, ABG+86] describe OSKA issues in further detail.

3 An OS Transport System Architecture Taxonomy

This section classifies OS Transport System Architectures (OSTSAs) by their *OS Kernel Architecture (OSKA)* dimensions, *OS Protocol Architecture (OSPA)* dimensions, and *software quality* dimensions. Table 1 depicts the taxonomy used to classify OSTSAs. Note that the OS Session Architecture (OSSA) and OS Network Application Programmatic Interface (OSNAPI) levels are not included in the taxonomy. This is because this paper focuses primarily on the OSPA and OSKA services that support the computing requirements of multiple protocol graphs on source and destination host machines. The OSNAPI and OSSA services, on the other hand, are primarily concerned with managing the system call interface to OSPA and OSKA services, and performing the end-to-end computing aspects of network protocols.

3.1 OS Kernel Architecture Dimensions

The OS Kernel Architecture (OSKA) provides services such as process management, virtual memory, and timer mechanisms. These services are employed by user application programs and other parts

[11] In a message-passing kernel, OSKA IPC is used to exchange messages between local processes [ABG+86]. These messages (and memory management scheme used to implement them) differ from the OSPA messages that are encapsulated and de-encapsulated as they move up and down a protocol graph (OSPA messages are described in Section 3.2.1).

Categories	Dimensions	Subdimensions	Alternatives
OS Kernel Architecture Dimensions	Process Architecture	(1) Concurrency Models	single-threaded, LWP, HWP, coroutines
		(2) Proc. Arch. Models	vertical, horizontal, hybrid
		(3) Parallelism Models	layer, directional, connectional, message, task
	Event Management	(1) Timing Relations	relative, absolute
		(2) Search Structure	array, linked list, heap
		(3) Event Notification	message passing, function call
	Virtual Memory Remapping		none, bi-directional, outgoing-only, incoming-only
OS Protocol Architecture Dimensions	Message Buffering		list-based, graph-based
	Multiplexing and Demultiplexing	(1) Synchronization	synchronous, asynchronous, hybrid
		(2) Layering	layered, non-layered
		(3) Search Method	sequential-search, hashing, indexing
		(4) Caching	none, single-item, multiple-item
	Layer-to-Layer Flow Control		per-queue, per-process
Software Quality Dimensions	Modularity	(1) Interface Uniformity	uniform, non-uniform
		(2) Data Coupling	low coupling, high coupling
	Flexibility and Extensibility	(1) Protocol Families	multiple, single
		(2) Configuration Time	static, dynamic
		(3) Composition Order	static, LIFO, arbitrary
		(4) Composition Typing	typed, untyped
		(5) OSPA Location	kernel-space, user-space, off-board

Table 1: OSTSA Taxonomy Template

of the operating system (such as the file subsystem and the OS Protocol Architecture (OSPA)). The OSPA is built on top of OSKA services that implement the *process architecture*, *event management*, and *virtual memory remapping* (which uses copy-on-write optimizations to reduce memory-to-memory copying overhead). Each of these dimensions is described below.

3.1.1 The Process Architecture Dimension

OSKA processes are fundamental operating system abstractions. A process consists of a collection of resources, along with one or more threads of control [ABG+86]. Process resources include virtual memory, CPU(s), file and device descriptors, access rights to other OS resources, etc. Threads of control act as separate instruction pointers within a single virtual address space.[12] Threads maintain state information (such as a stack of subroutine call activation records) that represents a program in execution. This state information allows processes and threads to be transparently suspended and resumed by the OSKA scheduler.

A *process architecture* is a framework that coordinates the independent execution of multiple OSKA processes in support of OSTSA protocol processing activities. This framework strongly impacts the performance of an OSTSA. In addition, it also influences the complexity of OSTSA software development. An effective process architecture makes it easier to design, implement, and modify both OSTSAs and network protocols without unduly sacrificing efficiency.

The OSNAPI, OSSA, and OSPA levels perform their services within the context of one or more cooperating OSKA processes. For example, multiple network connections may concurrently transmit

[12]The traditional BSD and System V UNIX OS process only contains a single thread of control. The *x*-kernel and Conduit, on the other hand, use multi-threaded processes.

and receive messages between peer session entities. Furthermore, within a given session, multiple protocol processing activities may run concurrently. For example, checksums may be computed in parallel with locating a session control block and computing round-trip time estimations.

The following section examines several dimensions of process architectures. It describes three different concurrent programming abstractions, compares and contrasts *horizontal* and *vertical* process architecture models, and discusses several ways to map process architectures onto multi-processors.

(1) Concurrent Programming Abstractions: Using separate OSKA processes to program concurrent threads of control is generally simpler than trying to explicitly synchronize and schedule multiple activities "by hand" (*i.e.*, outside the OSKA process architecture) [BA90]. However, to support concurrent protocol processing efficiently, the OSKA must minimize the overhead of preempting, rescheduling, and synchronizing executing processes and serializing access to shared resources.

Several concurrent programming abstractions that form the basis for many OSTSA process architectures include *heavy-weight processes*, *light-weight processes*, and *coroutines* [TRG+87]. Each abstraction entails different types of performance overhead and allows different levels of programmer control over process management activities like scheduling and synchronization.

- **Heavy-Weight Processes:** Heavy-weight processes (HWPs) reside in separate virtual address spaces within the OS kernel. Synchronizing, scheduling, and sending messages between HWPs typically requires a context switch. Context switching is a relatively expensive operation, since it usually requires copying registers and data between main memory and secondary storage, as well as flushing pipelines and invalidating virtual memory "translation lookaside buffer" caches.

- **Light-Weight Processes:** Light-weight processes (LWPs) are often referred to as a *threads*. Unlike HWPs, multiple LWPs usually *share* a virtual address space. This sharing reduces the overhead of thread creation, synchronization, and scheduling, since switching control between LWPs is less time-consuming than performing a context switch between several HWPs.

- **Coroutines:** In a coroutine model, the programmer, rather than the OSKA scheduler, explicitly chooses the next coroutine to run at some *synchronization point*.[13] With coroutines, the programmer has the flexibility to schedule processes in any desired manner. The programmer also has the responsibility, however, to handle all the details of scheduling, notably avoiding starvation and deadlock. Furthermore, coroutines only support "interleaved execution." This allow only one process to run at a given time, thereby limiting the benefits of parallel processing.

In order to produce efficient OSTSAs, it is important to match the design of the process architecture with the appropriate concurrent programming abstraction. In particular, it is essential to minimize context switching overhead [HP91].

(2) Different Process Architecture Models: As shown in Figure 5, there are two basic process architecture models: *horizontal* and *vertical*.[14] These architectures are logically equivalent. In other words, it is possible to implement the same protocol families (*e.g.*, TCP/IP, OSI, etc.) with either model. Important differences between the horizontal and vertical approaches involve *performance*

[13] For example, synchronization points occur when coroutine C_1 must "suspend" its activities to allow coroutine C_2 to execute its code. At some later point, coroutine C_2 may "resume" control to coroutine C_1.

[14] Different authors use these two terms in different ways. For example, [Haa91] uses the terms in a nearly opposite sense to describe the HOPS (horizontally oriented protocols) architecture. Haas defines a "vertical" architecture as one corresponding to a conventionally layered protocol graph such as the ISO OSI reference model (*i.e.*, what we are calling a "horizontal" architecture). The primary difference between the two usages of these terms stems from whether one chooses to focus on the layering "cuts" themselves or the components that results from these cuts. Our use of the horizontal/vertical terminology is consistent with [Cla85, Atk88, OP90a].

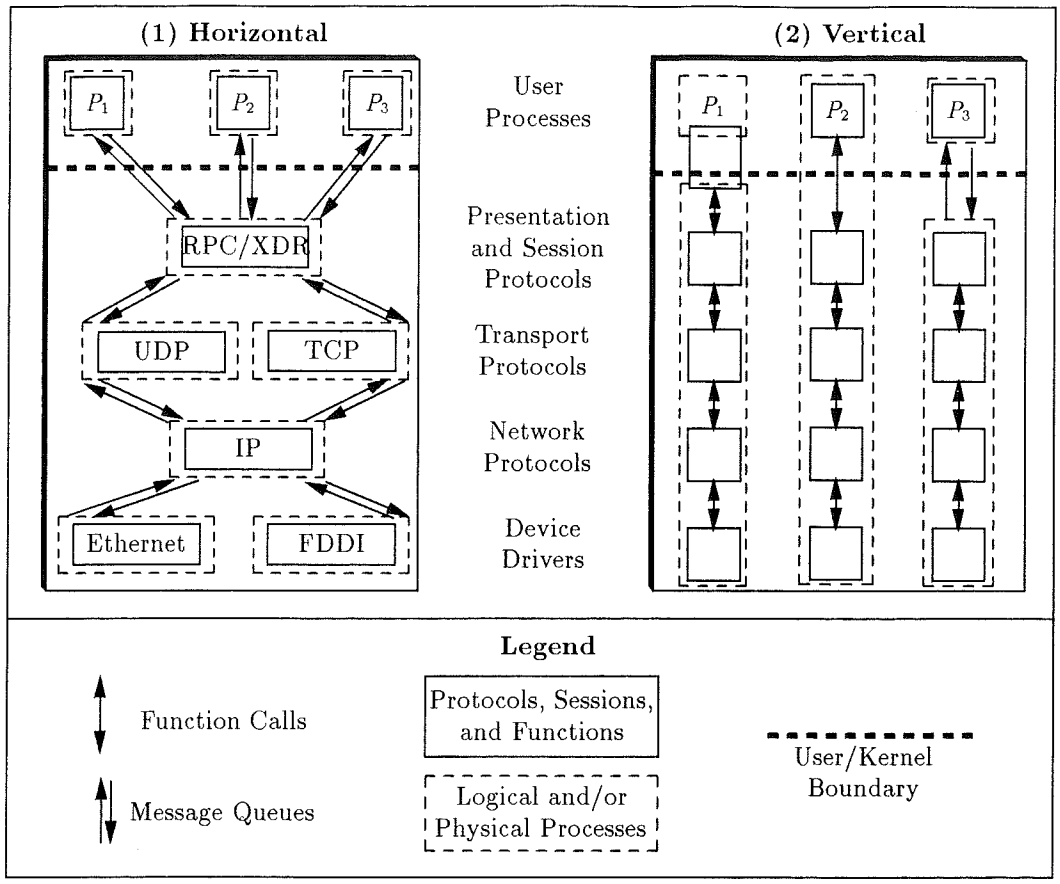

Figure 5: Horizontal and Vertical Process Architectures

(*e.g.*, process management, context switching, and messages-passing overhead) and *software design and implementation complexity* (which is related to the OSKA concurrent programming abstraction, *e.g.*, it is often easier to write complicate concurrent programs with light-weight processes, as opposed to coroutines).

The process architecture model is orthogonal to multi-processor support, *i.e.*, either vertical or horizontal process architectures may be implemented with single- or multi-threaded uni-processors or multi-processors. On multi-processors, separate processes may execute in parallel, although the extent to which separate processes actually run in parallel is constrained by synchronization and scheduling overhead. On uni-processor computers, some form of "time-slicing" may be used to provide logical (rather than physical) concurrency.

• **Horizontal Process Architectures:** Horizontal process architectures correspond closely to many layered protocol family specifications [Atk88]. Figure 5 (1) illustrates a hypothetical horizontal process architecture where user processes P_1, P_2, and P_3 exchange messages with Sun RPC/XDR (which, in turn, runs on top of the TCP, UDP, and IP protocols). In this model, each protocol layer is encapsulated in one or more light-weight or heavy-weight processes[15] that function as a pipeline for incoming and outgoing message. Messages flow between the processes as the result of multiplexing

[15] Note that a protocol running in a heavy-weight process typically resides in its own separate address space.

and demultiplexing operations. Each protocol layer processes the messages sent to it and then places them in a message queue for the next layer in the protocol graph. To improve performance, messages must be moved between processes with a minimal amount of memory-to-memory copying (which can be difficult if there is no global shared memory, as occurs with separate processing elements in a transputer architecture [Zit89]).

Horizontal process architectures have several advantages. First, the process architecture corresponds closely to layered protocol specifications like the ISO OSI [Bla91] and Internet [Com91a] reference models. This makes it relatively straight-forward to design and implement protocols in a horizontal architecture [Atk88]. Second, each protocol component manages its active sessions within a single process address space. This organization reduces the synchronization required to handle multiple messages bound for the same active session, since only one process controls a given protocol component's internal data structures.[16]

However, horizontal process architectures have several significant disadvantages. For example, the amount of available parallelism is rather limited. This is due to the fact that most major protocol suites specify only a small number of protocol layers. For example, the Internet reference model has only four primary layers (*e.g.*, data link, network, transport, application) and the OSI reference model has just seven layers (*e.g.*, physical, data link, network, transport, session, presentation, application). Therefore, the amount of available parallelism is rather limited if there is only a one-to-one correspondence between processes and protocol components. A more severe disadvantage stems from the context switching, scheduling, synchronization overhead associated with horizontal process architectures. Messages flowing up and down between protocol layers incur a large amount of interprocess communication (IPC) overhead, since in a horizontal architecture, each protocol layer corresponds to one or more processes. IPC overhead between protocol layers strongly influences overall system performance and throughput [Cla85]. Due to this overhead, most high-performance OSTSAs avoid highly-layered horizontal process architectures [CT90].

• **Vertical Process Architectures:** Vertical process architectures represent a more recent OSKA structuring approach for OSTSAs [Cla85, Atk88, JSB90]. Figure 5 (2) illustrates one hypothetical vertical process architecture that implements the same protocol graph as Figure 5 (1).[17] Unlike the horizontal process architecture example, in this example the OSKA associates a separate process to each incoming and outgoing message [HP91]. Each process escorts its message through the protocol graph, delivering it "down" to a network interface or "up" to a user application process. Since each process resides in its own address space, messages flow through active protocol sessions via synchronous subroutine calls rather than asynchronous IPC mechanisms (such as the message queues used by the horizontal model).

Figure 5 (2) also illustrates three different ways that user applications exchange information with network protocols. Process P_1 interacts with an OSNAPI data queueing endpoint (similar to System V STREAMS and BSD UNIX), process P_2 uses *upcalls* running in the same process (similar to the *x*-kernel approach), and process P_3 uses asynchronous message queues.

Vertical process architectures have several advantages compared to horizontal approaches. First, there is greater potential for exploiting available parallelism, since every arriving and departing message is associated with its own process [JSB90]. Increased parallelism also enables improved processor load balancing (which potentially improves overall OSTSA throughput).[18] For example, if processes are carefully implemented on a multi-processor, each incoming message may be dispatched

[16] One consequence of this architectural design is that multiple messages are serialized at each protocol component.

[17] Note, there are other ways to organize a vertical process architecture, one of which (connectional parallelism) is described below.

[18] The actual benefit from load balancing depends heavily on its interaction with *cache affinity* [VZ91] effects, which involve the interaction between scheduling policies and instruction and data caches on shared memory multiprocessors.

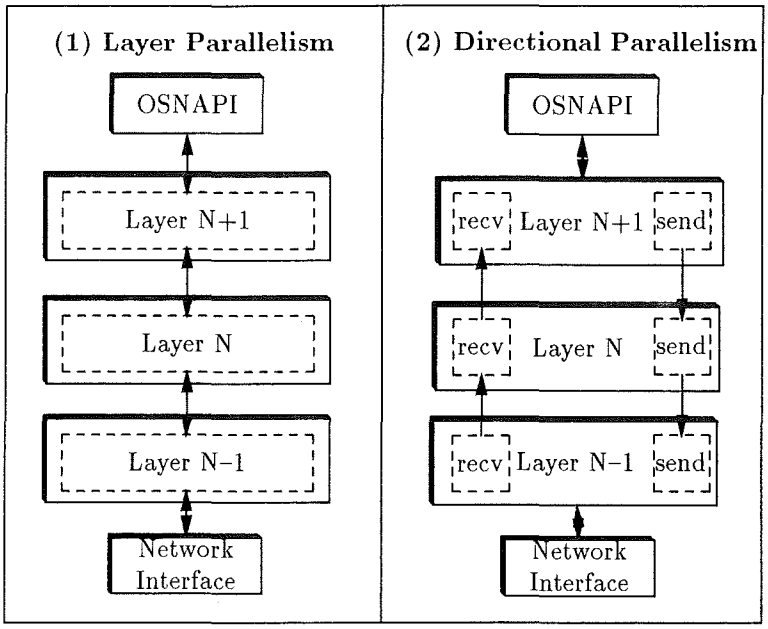

Figure 6: Models of Parallelism for Horizontal Process Architecture

to an available processing element. Second, context switches are not required to multiplex and demultiplex messages between protocol layers. Since protocol layers reside in the same address space, synchronous subroutine calls are used to communicate between the layers. This is substantially faster than performing IPC with asynchronous message queues, since exchanging messages between protocol layers does not incur context switch overhead [HP91]. Finally, the vertical process architecture does not impose a total ordering on messages bound for the same connection. This is an advantage for network protocols that require only partial orderings between messages (*e.g.*, the Psync IPC protocol [PBS89] which uses partial orderings to implement "many-to-many" group communication efficiently) or that utilize *application level framing* [CT90] (which is a design principle that maintains application data unit boundaries throughout lower-layer protocol processing stages).

Vertical process architectures also have several disadvantages. First, performance may suffer if the OSKA cannot efficiently associate an process with each message. This is particularly problematic when overall system communication loads are very high (*i.e.*, a large number of messages are arriving and departing). One potential solution for this problem is to cache processes in a "process pool" and recycle them for subsequent messages [HP91]. However, these cached processes may sit idle when overall system communication activity is light, thereby "tying up" OS resources like memory buffers and process table entries (which may also be needed by other OS subsystems). Second, increased synchronization overhead and memory contention may occur when complex interactions occur between messages and sessions at the receiver. For example, multiple messages bound for the same higher-layer sessions (*e.g.*, as the result of TCP segmentation or IP message fragmentation at the sender) must coordinate and synchronize in order to share session state information correctly, efficiently, and consistently between multiple processes.

(3) Mapping Process Architectures onto Multi-Processors: Several forms of multi-processing have been suggested to develop OSTSAs that effectively support gigabit networks [Zit91, JSB90, Haa91, CG91, GKWW89, HEHK92]. This section examines a number of approaches for mapping horizontal and vertical process architectures onto multiple *processing elements* (PEs).

To improve the benefits from multi-processing, a parallel implementation of a process architecture should meet several criteria [JSB90]. First, the process architecture should be amenable to significant levels of parallelization. For example, a process architecture that only utilizes two PEs is not as likely to scale up as well as one that effectively utilizes dozens of PEs (all other factors held equal). Second, overall throughput will suffer, if too much time is spent coordinating activities between PEs. This implies that an effective multi-processor architecture should strive to minimize interprocess communication and synchronization overhead, while taking advantage of cache affinity properties. Finally, processing loads should be carefully distributed between the multiple PEs to reduce bottlenecks and "hotspots."

Five models of process architecture parallelism, (1) *layer parallelism*, (2) *directional parallelism*, (3) *message parallelism*, (4) *connectional parallelism*, and (5) *task parallelism*, are illustrated in Figure 6, 7, and 8 and described below. These five models are differentiated by their *granularity*, ranging from "coarse-grained" to "fine-grained." Granularity is determined both by the size of the tasks associated with each PE and the number of PEs involved. In general, coarse-grained approaches (*e.g.*, *directional parallelism* and *connectional parallelism*) are simpler to design and implement than the finer-grain approaches (*e.g.*, *message parallelism* and *task parallelism*), since there is less interaction between the PEs. However, coarse-grain approaches are also less scalable in their degree of potential parallelism. Determining the conditions under which a particular model is more scalable and efficient remains an open research question.

- **Layer Parallelism:** Layer parallelism is a straight-forward implementation of a horizontal process architecture. In this approach (shown in Figure 6 (1)), a PE is associated with each layer in the protocol graph. Messages flow through the layers in a pipeline fashion. The primary disadvantages are that potential parallelism is limited to the number of protocol layers and there is typically high overhead to move between layers.

- **Directional Parallelism:** Directional parallelism (shown in Figure 6 (2)) is similar to layer parallelism, though it dedicates two PEs per-protocol layer, one for sending outgoing messages and another for receiving incoming messages. This model is also relatively easy to conceptualize and design, though it provides only a multiplicative increase in parallelism compared to layer parallelism. Moreover, unless protocol input and output operations are relatively independent, communication between the sending PE and receiving PE in a protocol component may become a source of overhead. For example, protocols such as TCP, where acknowledgments for incoming segments are "piggy-backed" on outgoing data and control messages, require communication and cooperation between sender and receiver [GKWW89]. Finally, as with layer parallelism, directional parallelism does not facilitate PE load balancing, since PEs are dedicated to specific protocol processing layers.

- **Message Parallelism:** Figure 7 (1) depicts message parallelism, which involves associating a separate PE with each incoming or outgoing message. Compared with the previous two parallelism models, the advantages of this approach are (1) the degree of parallelism is potentially quite high (being a function of the number of messages, minus the synchronization overhead and cache affinity effects), (2) communication overhead decreases (since moving between protocol layers may not involve a context switch), and (3) messages may be more evenly balanced between PEs. However, this model appears somewhat easier to conceptualize than to implement, due to the complexity and overhead of synchronizing messages bound for the same higher-layer network connection. For example, the synchronization overhead resulting from obtaining locks required to gain exclusive access to shared

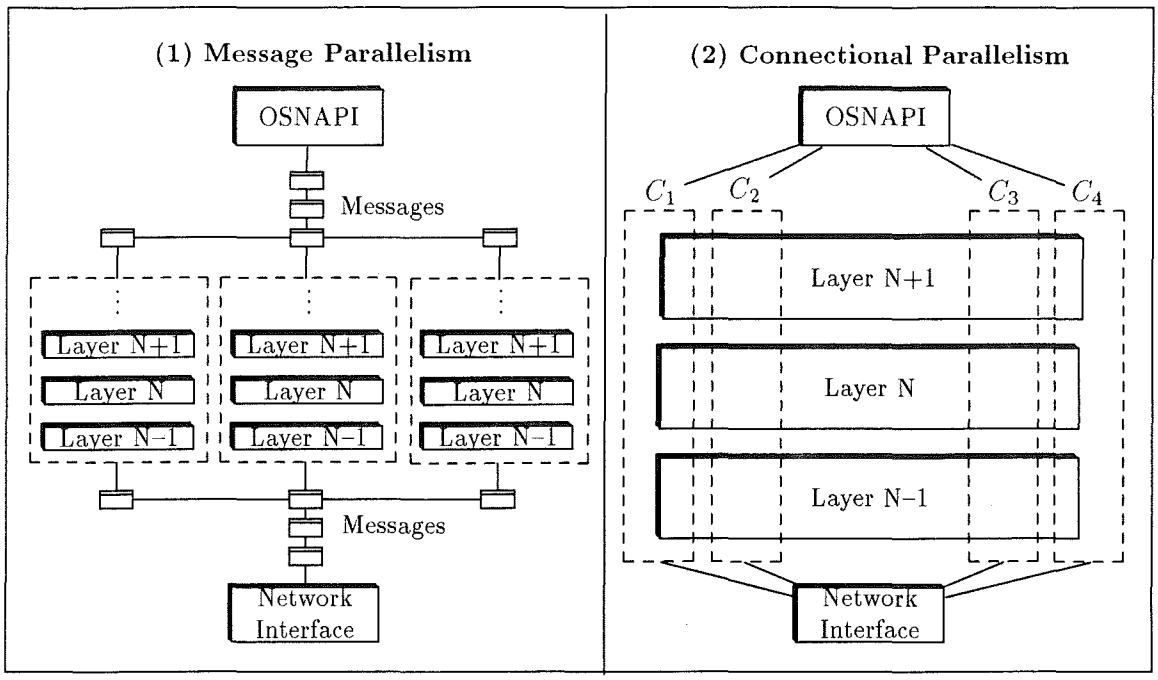

Figure 7: Models of Parallelism for Vertical Process Architectures

resources (*e.g.*, memory buffers or session control blocks) may become a bottleneck when joining together TCP segments bound for the same higher-layer session. Moreover, overhead may also occur from factors such as shared-memory bus contention [JSB90].

• **Connectional Parallelism:** Connectional parallelism dedicates a separate PE for each active connection. Figure 7 (2) illustrates this approach, where connections C_1, C_2, C_3, and C_4 are each associated with a separate process that is responsible for processing all messages addressed to that connection. This approach may be useful for servers that have several connections open simultaneously [HEHK92]. The degree of parallelism in this approach is a function of the number of active connections. One drawback is that it is difficult to balance PE loads. For example, a highly active connection might swamp its PE with excessive work, even though other PEs sit idle at inactive connections. Unlike directional parallelism, synchronization and communication overhead is relatively low (within a given connection).

• **Task Parallelism:** Task parallelism is an example of very "fine-grain" parallelism that applies multiple PEs to perform multiple tasks on a per-message or per-protocol basis. Figure 8 illustrates an example of task parallelism where multiple processes perform or coordinate several operations in parallel on a message. These operations include computing checksums (usually performed in hardware to improve efficiency), decoding an address field in a message header, searching various tables for protocol and session control blocks, and computing round-trip time estimates. Since most protocol processing tasks appear to have large amounts of interdependency, it may be difficult to eliminate memory contention and synchronization overhead. One proposed strategy for alleviating the overhead from these interdependencies is to pipeline the message processing.[Zit89, GKWW89]

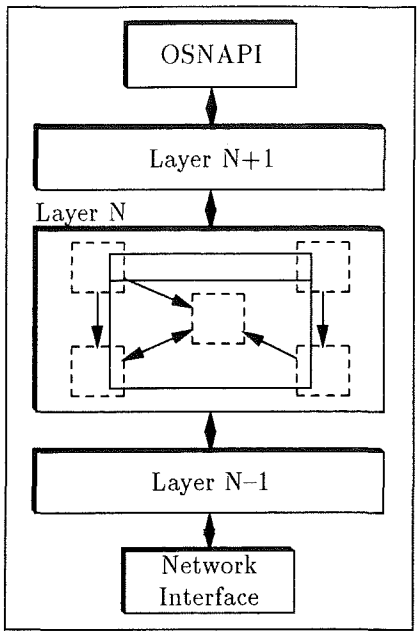

Figure 8: Task Parallelism

Figure 9 plots the relationship between the process architecture models and the five parallelism models. Layer parallelism appears to be the most coarse-grain approach and task parallelism appears to be the most fine-grain approach. Note that both layer and directional models have a fixed amount of parallelism (*i.e.*, corresponding to the number of protocol layers), whereas the parallelism available in the message and connectional models varies according to the number of messages and connections, respectively. Finally, it may also be possible to combine these models, forming more complicated architectures such as HOPS (horizontally oriented protocols) [Haa91]. Sections 3.2.2 and 3.2.3 examine the relationship between the process architecture and other OSTSA dimensions such multiplexing, demultiplexing, and flow control.

3.1.2 The Event Management Dimension

The OSKA event manager provides timing-related services used by both user applications and components in other OSTSA levels (*i.e.*, OSNAPI, OSSA, and OSPA). A typical event manager interface is modeled as an Abstract Data Type (ADT) that provides three basic operations: (1) registering a subroutine that will execute at some user-specified time in the future, (2) canceling a previously registered subroutine, and (3) asynchronously invoking a registered subroutine when its "time-to-execute" occurs. In order to support real-time applications, it is important to perform all three of these services efficiently, with small amounts of variance (even when there are a large number of registered subroutines). These services are built on top of a hardware-interrupt-driven clock mechanism. On each clock tick the event manager checks whether it is time to execute any of the scheduled events. If one or more events must be run, the event manager invokes the associated subroutines.

At the OSSA level, many network protocols perform time-related operations on active and inactive network connections, and these operations use the event management services provided by the OSKA. Some of these operations are driven by timers that are set or canceled in response to protocol-related events. For example, when a TCP segment is sent, a retransmission handler subroutine is registered with the event manager. The time to execute is based on a time interval calculated from the TCP

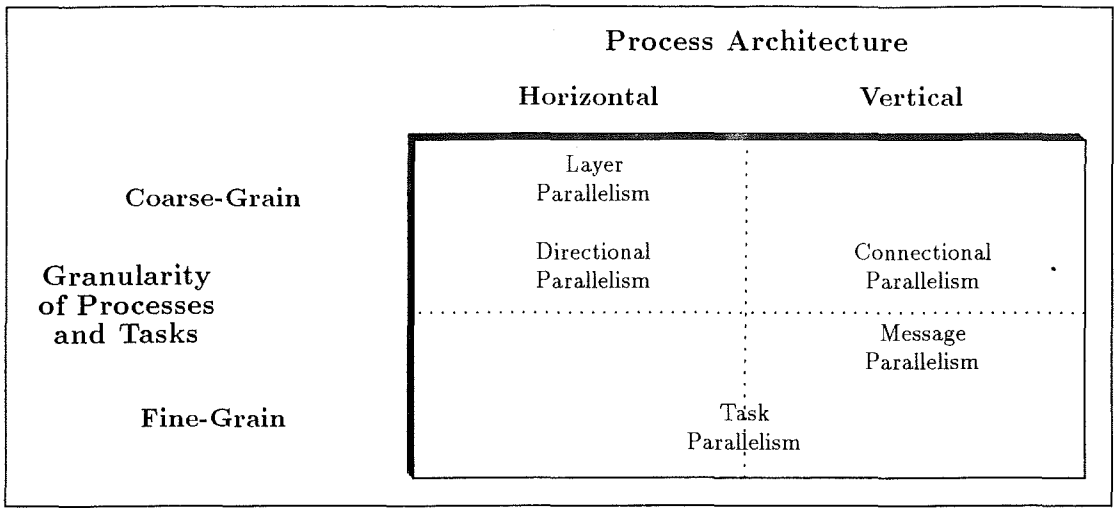

Figure 9: Relationship Between Process Architecture and Parallelism Granularity

round-trip estimate for that connection. When the timer expires, the event manager calls the handler, which then retransmits the segment. If an acknowledgement for the message arrives before the timer expires, on the other hand, the handler is canceled.

Timers can also be scheduled to run periodically, in which case they do not correspond directly to any particular protocol-related event. For example, the SNR transport protocol uses a *periodic* interval timer to synchronize sender and receiver state information. In this scheme, a receiver does not explicitly acknowledge the sender when it receives a message, but instead waits for a periodic interval timer to expire. At this point, the previously registered handler exchanges the receiver's complete state information with the sender [NRS90].

Example OSKA event management implementations include *delta lists* [Com91b], *timing wheels* [VL87], and heap-based [BL88] and linked list-based [LMKQ89] *callout queues*. The following three primary dimensions classify these different event management mechanisms.

(1) ADT Implementation: There are several common strategies for implementing the event management Abstract Data Type (ADT). A simple approach is to sort the events by their "time-to-execute" value and store them in an array [Bac86]. A variant on this approach replaces the array with a sorted linked list (which reduces the overhead of adding or deleting an arbitrary event handler) [Com91b]. A third approach uses a heap-based priority queue [BL88]. Using a heap instead of a sorted list or array reduces the time complexity for inserting or deleting an entry from $O(n)$ to $O(\lg n)$ time. This can save a significant amount of time in a large system where many devices use the event manager (*e.g.*, terminals and network connections).

(2) Time Relationships: A second aspect of event management involves time relationships, *i.e.*, whether *relative* or *absolute* timing is used to sequence events. Relative time is typically used with a sorted array or sorted linked list ADT. Every item in the array or list corresponds to an event scheduled to occur in the future. Because each item's time is stored relative to the previous item, the event manager only needs to examine the first element in the array or list on every clock tick to determine if it should execute the event handler. On the other hand, heap-based approaches use absolute time, due to the operations required to maintain a heap.

(3) Event Notification Mechanism: When a timer expires, the event management mechanism either calls a registered subroutine with its associated argument [BL88] or it may send a control message to a port via a message queue [Com91b] (see Section 4.1.5 for additional details).

3.1.3 The Virtual Memory Remapping Dimension

Regardless of the process architecture, achieving efficient network protocol performance requires minimizing the amount of memory-to-memory copying performed throughout the OSTSA [WM87]. In general, memory copying provide an upper bound on user application throughput [CT90]. Choosing an efficient message management mechanism is one method for reducing copying overhead (see Section 3.2.1 below). A related approach uses OSKA virtual memory facilities to avoid expensive data copying. For example, in situations where data copies would ordinarily be performed[19], the OSKA remaps virtual memory pages instead, marking them "copy-on-write." Page remapping is particularly useful for transferring large quantities of data between separate address spaces [YTR+87].

Several complications arise with page remapping schemes that make them difficult to implement in practice. First, most page remapping schemes require placing data in contiguous buffers that begin on page boundaries. Ensuring this alignment restriction may be complicated by other protocol operations and options such as message de-encapsulation (*i.e.*, stripping headers and trailers as messages migrate up a protocol graph), presentation layer expansion [CT90] (*e.g.*, uncompressing or decrypting an incoming message), and variable-size header options. Certain versions of BSD UNIX support a "trailer option" that places variable-size protocol headers at the end of a message, so that the fixed-size data portion comes first. This technique facilitates remapping (or at least minimizes copying) by allowing incoming messages to be aligned on page boundaries. Second, remapping may not be useful if the remapped page is immediately written upon, since a separate copy must be made anyway [LMKQ89]. Finally, if messages are small, there may be more overhead in remapping them (*e.g.*, adjusting page table entries, invalidating translation-lookaside buffers, etc.) compared with simply copying them in the first place.

3.2 OS Protocol Architecture Dimensions

The OS Protocol Architecture (OSPA) supports intra-protocol (*e.g.*, session graph management, message management) and inter-protocol (*e.g.*, protocol graph management, layer-to-layer flow control, multiplexing and demultiplexing of messages) services that are common to most network protocols. OSPA services pertain primarily to network protocols and distributed applications (as opposed to OSKA services that are also utilized by most other OS subsystems).

3.2.1 The Message Management Dimension

Various types of messages are used throughout the OSTSA to exchange data and control information between local and remote peer entities. Some standard network message management operations include storing messages in buffers as they are received from network interfaces, prepending and/or stripping headers and appending and/or stripping trailers from messages as they flow through various protocol layers, storing messages into buffers for transmission or retransmission, fragmenting and reassembling messages, and reordering messages received out-of-sequence [JSB90].

An effective message manager for networking applications must fulfill several general requirements. First, it must efficiently support both fixed-size and variable-size allocations and deallocations of

[19]Transferring messages from kernel-space to user-space is a common OSNAPI-level operation that often involves memory-to-memory copies [OAHP90].

memory. Network traffic tends to have a bi-modal distribution of sizes, either large messages (*e.g.*, for bulk data transfer) or small messages (*e.g.*, for remote login and voice applications). Second, it must support *protocol encapsulation.* Encapsulation occurs as messages move up and down a protocol graph; it involves adding and deleting both headers and trailers to the beginning and end of a message, respectively. Third, a message manager must also support *fragmentation* and *reassembly.* Finally, it must implement these operations with a minimal amount of data copying.

As mentioned in Section 3.1.3, memory-to-memory copying is a significant source of OSTSA overhead. Naive implementations that physically copy messages between each layer are far too expensive. Therefore, message management schemes are optimized to minimize data copying, using techniques such as *buffer-cut-through* (passing buffers by reference through multiple protocol layers [WM89, ZS90]) and *copy-on-write* schemes (that use lazy evaluation, reference counting, and buffer-sharing to avoid making unnecessary copies [OAHP90]).[20]

Message management schemes are often tuned to work efficiently for different message sizes. For example, certain schemes are well suited for small- or large-size messages, but not for medium-size messages. In particular, the BSD message management facility divides its buffers (called *mbufs* into small (128 byte) and large (1,024 byte) blocks. This leads to non-uniform performance as incoming and outgoing messages vary between small and large mbuf sizes [HMPT89]. Different schemes also vary in their level of support for minimizing data copying and data sharing. For instance, a standard message management scheme (*e.g.*, used by BSD UNIX and System V) chains multiple pieces of a message together to form a linked-list of message fragments. Adding data to the front or rear of the list only involves relinking pointers, and does not require any data copying. An alternative approach uses a Directed-Acyclic-Graph (DAG)-based message data structure [OAHP90]. This method provides better support for data sharing between protocol layers, since DAGs allow multiple "parents" to share a single "child."

3.2.2 The Multiplexing and Demultiplexing Dimension

Multiplexing and demultiplexing are mechanisms used to route messages between sessions in one or more adjacent protocol layers. Multiplexing is typically performed at the sender's end of a network connection. It directs outgoing messages from some number of higher-layer sessions onto a smaller number (usually one) of lower-layer sessions [Ten89]. Demultiplexing performs the inverse task on the receiver's end by directing incoming messages up to their associated sessions. Note that multiplexing and demultiplexing are orthogonal to data copying. In other words, depending on the message management scheme, messages may not require memory-to-memory data copying as they move up and down through the protocol layers [OAHP90].

In general, demultiplexing is more complicated than multiplexing as the result of several factors. First, the sender has knowledge about the entire transfer state [CT90] (*e.g.*, the destination address[21] of the messages and which network interface to use). For connection-oriented services, this information may be precomputed at connection establishment time and reused for subsequent messages destined for the same address. On the other hand, when a network controller receives an incoming message, it must inspect the message header and perform a lookup operation to determine which higher-layer protocol should receive the message. This demultiplexing operation may occur several times enroute from network controller to user process.[22] Second, demultiplexing often requires

[20] These schemes may be combined with the virtual memory remapping optimizations described in Section 3.1.3.

[21] Addresses indicate which local and/or remote process(es) should receive a particular message. Examples include port numbers, connection identifiers, and Internet IP addresses.

[22] For example, IP messages are demultiplexed on a header field indicating whether the message is bound for TCP, UDP, or some other higher-layer protocol; likewise, these higher-layer protocols may demultiplex further up to an

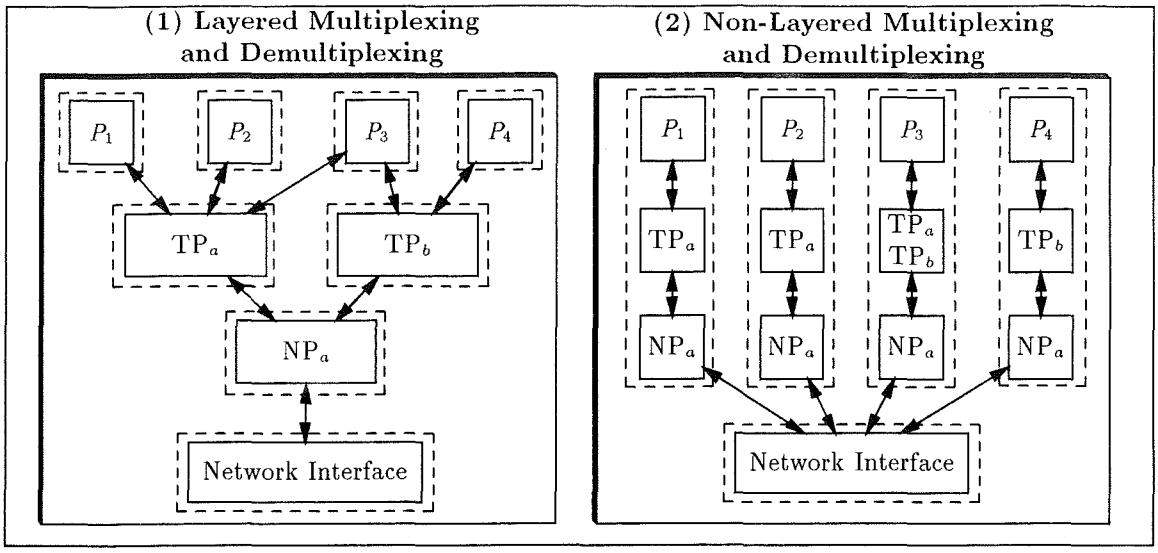

Figure 10: Layered and Non-Layered Multiplexing and Demultiplexing

dynamically allocating data structures (*e.g.*, in order to deliver messages addressed to "passively-opened" connections). Finally, depending on the process architecture, these demultiplexing activities may exhibit high synchronization and context switching overhead, since multiple processes may need to be *awakened*, scheduled, and executed. These factors help explain why receivers, rather than senders, are often performance bottlenecks in distributed systems [Hol91].

Four important multiplexing and demultiplexing dimensions are *synchronous vs asynchronous*, *layered vs non-layered*, and *different search methods*, and *different caching strategies.*

(1) Synchronous vs Asynchronous: Multiplexing and demultiplexing may be either synchronous or asynchronous. Whichever method is chosen strongly relates to whether the OSKA uses a horizontal or vertical process architecture (see Figure 5). For example, vertical process architectures (like the *x*-kernel) typically use synchronous multiplexing and demultiplexing. Since each vertical process resides in its own separate address space *upcalls* and subroutine calls may be used to transfer messages up and down the protocol graph. In this case, the demultiplexing operation simply determines which higher-layer protocol to invoke; calls from lower-layer protocols block until higher layers complete their protocol processing.

On the other hand, horizontal process architectures (*e.g.*, Xinu or System V STREAMS) often use asynchronous multiplexing and demultiplexing that pass messages up and down the protocol graph *without* blocking the sender. This approach requires message queues to buffer data between layers and may also require additional context switching.

(2) Layered and Non-Layered Multiplexing and Demultiplexing: Multiplexing and demultiplexing route messages between OSPA protocols and sessions. In *layered* schemes (shown in Figure 10 (1)), this routing may occur multiple times as messages traverse up or down the protocol graph. This approach differs from *non-layered* multiplexing and demultiplexing (shown in Figure 10 (2)), where the routing decision is performed only once, usually at the lowest-layer of the protocol graph, *e.g.*, at the network interface layer. The choice between layered and non-layered multiplexing and demultiplexing has an important impact on both OSTSA performance and modularity.

associated user process.

Using layered multiplexing and demultiplexing has several advantages [Ten89]. First, it promotes modularity, since services offered at one layer may be developed independently from other layers. Second, it helps conserve resources (*e.g.*, virtual circuits) by sharing them among higher layer sessions.[23] Finally, layered approaches are useful for coordinating different simultaneous multimedia applications (*e.g.*, synchronized voice and video streams), since messages are forced to synchronize at each Service Access Point (SAP) boundary.

The main disadvantages of layered multiplexing and demultiplexing result from the additional overhead of performing multiple routing decisions. For example, depending on the OSTSA process architecture, multiple levels of demultiplexing may lead to high context switching and synchronization overhead. Layering interactions also often increase *jitter*, which is detrimental to the performance of many delay- and jitter-sensitive multimedia applications. Some researchers believe that this overhead outweighs the benefits described in the previous paragraph [Ten89].

Therefore, non-layered multiplexing and demultiplexing has been proposed as an alternative. Non-layered approaches are beneficial for several reasons. First, they decrease contention from network connections that are transmitting and receiving from the same protocol component or protocol layer, because there is less competition for the same lower-layer SAPs [Ten89]. Second, in a horizontal process architecture, using a non-layered approach will reduce the number of processes and therefore decrease the total amount of context switching overhead.

However, there are several disadvantages to using a non-layered approach. First, it expands the *degree* of demultiplexing at the lowest layer. This violates certain protocol layering assumptions, since the lowest layer must be able to determine and demultiplex on session identifiers that occur several logical layers above it in a protocol graph. Second, it increases the number of sessions within every intermediate protocol layer, since these sessions are replicated and not shared [Ten89]. Finally, they encourage monolithic, special-purpose implementations that are difficult to maintain and extend [CT90].

(3) Search Mechanisms: Implement a multiplexing and demultiplexing scheme typically involves some form of searching. For example, BSD's TCP implementation searches a list of control blocks to demultiplex incoming messages to their appropriate connection session. The search key is known as an *external identifier* (*e.g.* network addresses, port numbers, and type-of-service field); it is used to locate some internal capability (*e.g.*, pointers to session state information, protocol control blocks, and network interfaces). Several popular search algorithms include *direct indexing* (*e.g.*, using a *connection identifier*), *sequential-search*, and *hashing*.

Certain transport protocols (*e.g.*, TP4 and VMTP) have connection identifiers that may be used to decrease demultiplexing overhead. For example, these identifiers may be computed at connection establishment time. This greatly simplifies the demultiplexing operation by directly indexing into the associated control block, rather than searching on keys in the form of the <*source addr, source port, destination port*> tuple used to identify TCP and UDP associations. If a particular protocol does not support connection identifiers, sequential-search or hashing may be used instead. Searching a linked list or table sequentially is simple to implement, though it does not scale up well if there are a large number of items in the key's search space (*e.g.*, many open network connections). Therefore, some form of hashing (such as *bucket-chaining*) is often used if many search keys exist [HMPT89].

(4) Caching and List Reorganization: There are several optimizations available for the above search methods. Optimizations include using single- or multiple-item caches, along with list reorganization heuristics that move recently accessed control blocks to the front of the search list or bucket.

[23] For instance, This sharing is useful for leased-line communication links, where it is expensive to reestablish a dedicated virtual circuit for each transmitted message.

If applications form "message-trains" (where a sequence of back-to-back messages are destined for the same higher-level session), then a single-level control block cache is a relatively efficient, straightforward implementation [MD91]. On the other hand, single-level caching is not particularly efficient for applications that do not form message-trains.[24]

In general, the different search algorithms and optimizations have a significant impact on overall OSTSA and protocol performance. Hashing, combined with caching, produces a measurable improvement when searching large lists of control blocks (*e.g.*, representing the associated network connections) [HP91].

3.2.3 The Flow Control Dimension

Flow control is a mechanism used by a sender or receiver to regulate the rate of speed and amount of data that is being transmitted. Flow control is necessary due to resource limitations in an OSTSA implementation. In particular, an OSTSA will not dedicate an infinite amount of memory for servicing network connections.

There are two kinds of flow control (*peer-to-peer* and *layer-to-layer*) that correspond to the different OSTSA levels in which network communication occurs. Peer-to-peer flow control is applied on a per-connection basis at the OSSA level to avoid transmitting messages faster than the remote receiver is able to store and process them.[25] For example, at the transport layer, TCP uses a "sliding window" flow control algorithm to regulate the amount of data exchanged between two network connections.

Layer-to-layer flow control occurs both between and within other OSTSA levels. At the OSNAPI level, flow control is usually performed by blocking a user process that attempts to send and/or receive more data than the inner-level OSPA protocol and session components are capable of handling at that moment. Within the OSPA level, layer-to-layer flow control is used to prevent higher-layer protocols from flooding lower-layer protocols with more messages than they are equipped to process and/or buffer. Two general mechanisms for controlling layer-to-layer flow are *per-queue* flow control and *per-process* flow control. They are described in the bullets below.

- **Per-Queue Flow Control:** Flow control is often implemented by putting a limit on the number of messages or number of total bytes that are queued between or within adjacent protocol layers. For example, a horizontal process architecture (*e.g.*, System V STREAMS) places a limit on the size of the message queue used to pass information between adjacent protocol layers (or between the top-most protocol layer and an application program executing as a user process).

- **Per-Process Flow Control:** Flow control may also be performed on a per-process basis. For example, in a vertical process architecture (like the *x*-kernel), an incoming message is discarded if a light-weight process is not available to shepard it up the protocol graph.

3.3 Software Quality Dimensions

This section examines *modularity*, *flexibility*, and *extensibility*, which are three software quality dimensions related to the design and implementation of Operating System Transport System Architectures (OSTSAs). Although these quality dimensions are difficult to quantify precisely, they affect the correctness, performance, portability, maintainability, and reusability of OSTSA software.

[24] Note that when determining caching benefits, the "miss ratio" (*i.e.*, how many times the desired external identifier is *not* in the cache) represents only part of the overall demultiplexing cost. It is also important to consider how many list entries must be searched when a cache miss occurs. If search lists are long, the cost of a cache miss may be high.

[25] Note that network congestion may also force buffering of data at the OSSA level on the sender.

3.3.1 The Modularity Dimension

Modularity is a software quality dimension that promotes reusability, flexibility, and extensibility [Mey89]. In general, using a modular design aids the software construction process by reducing development and maintenance costs and increasing system quality. Modularity divides large, complex systems into smaller, intellectually manageable components [Wir71], and localizes the effects of specification changes and programmer errors to within well-defined modules[26] [Hen80]. Many software design methodologies emphasize modularity [Mye78, Par72, YC79].

In distributed systems, modularity is often manifested by layered designs and implementations. In fact, many research projects [Cla85, HP91, Zwe91, OP91, Sha91] propose different approaches for developing modular and efficient software architectures for distributed OSTSAs. A modular decomposition of an OSTSA separates system functionality into distinct components or layers. It is difficult to measure the degree of OSTSA modularity precisely. However, two empirical indicators of OSTSA modularity include *interface uniformity* [OP91] (*i.e.*, the uniformity of the service interfaces exported within and between software components and hierarchical layers) and the degree of *data coupling* [Mye78] between components. The following bullets discuss these indicators.

(1) Interface Uniformity: There are two general types of interface uniformity. First, modules that perform the same abstract services (*e.g.*, connection establishment, flow control, message management, etc.) possess high uniformity if their service interfaces remain the same regardless of their service implementations. Second, protocol components in a layered protocol graph may also exhibit uniformity at their Service Access Points (SAP) (which occur at the boundaries between protocol layers). In this case, highly uniform interfaces exist if all the SAPs in the protocol graph use the same service interface. For example, passing a message between the user-process-to-TCP-protocol SAP boundary would use the same service interface as passing a message between the TCP-protocol-to-IP-protocol SAP boundary.

Interface uniformity is an important ingredient for building reusable software components for network protocols. Lack of uniform interfaces makes it difficult to support *protocol substitution* (which is the ability to transparently interchange protocols that provide the same class of service [Bro88]). Furthermore, uniformity enhances simplicity [Com91b], and it is generally easier to reuse simple modules that possess standard interfaces [OP91].

(2) Data Coupling: A second modularity indicator involves measuring the degree of data coupling between software components in an OSTSA. Research suggests that high data coupling between software components is associated with higher defect rates and higher maintenance costs [HK81a]. Software analysis tools are available to measure data coupling within software systems. For example, *data binding* metrics [HB85, SB91, Sel88] are one technique for quantifying the data dependencies between software system modules. Another technique computes *information flow* metrics, which measure the degree of module *fan-in* and *fan-out* [HK81b].[27]

In general, highly-layered and highly-modularized OSTSAs (such as the Conduit framework and the *x*-kernel) possess high interface uniformity and low data coupling among their components and layers. This implies that modules in these systems minimize inter-module data dependencies and respect layering boundaries. For example, protocol and session objects in the *x*-kernel only reference

[26] A module is defined here as "a software component encapsulating the representation of some abstraction." Modules may be either stand-alone subroutines or abstract data types (ADTs), which are collections of related data structures and subroutines that directly update and/or retrieve data structure state information [HK81b].

[27] Fan-in measures the number of modules that pass data (via parameters or global variables) into a module. Fan-out measures the number of modules receiving data from a given module.

local variables, and only access lower-layer components via well-defined control interfaces [OP91]. Monolithic OSTSA decompositions, on the other hand, often de-emphasize or ignore layering boundaries. This is manifested by lower interface uniformity and higher data coupling.

There are several advantages to organizing and describing OSTSA software in a modular and layered manner. First, layering enables multiple outer-level OSTSA components (*e.g.*, user processes and network protocols) to share inner-level services [Fel90] (*e.g.*, process and virtual memory management, and layer-to-layer flow control mechanisms). Second, viewing OSTSAs as virtual machine levels with well-defined service interfaces enables transparent, incremental enhancement of communication services [Ten89]. Finally, modular designs generally improve the implementation, testing, and maintenance of software systems [Par72, Par79, PCW83]. This, in turn, reduces overall development effort, facilitates reuse of existing software components, and creates more flexible and extensible OSTSAs.

There are also several disadvantages of using modular and layered approaches for OSTSAs [Ten89, CWWS92]. A common criticism of layered implementations is that they introduce too much overhead, which prevents them from delivering high-speed network bandwidth to applications [Cla82]. This overhead typically results from several factors. First, modular and highly-layered OSTSAs incur significant inefficiencies if layering is not carefully structured with the process architecture [HP91, CWWS92]. In particular, in a horizontal process architecture, the context switching overhead may be so large that efficiently supporting a highly-layered protocol graph (where each layer is encapsulated in a separate process) is extremely difficult [BO91]. Second, modular systems incur a performance penalty by encapsulating OSTSA data structures and protocol state information behind abstract service interfaces. This approach requires protocol processing activities to use subroutine calls and parameter passing to access desired information. The overhead from encapsulation may be quite significant for protocol families like TCP/IP, where TCP protocol processing involves a strong dependency on IP protocol state information.[28] Monolithic implementations avoid this encapsulation overhead, since information hiding is either non-existent or not strictly followed.

Monolithic implementations are not *necessarily* always more efficient than modular ones, however. In fact, carefully designed modular architectures may actually "enhance" performance in several ways. First, modular OSTSA components are potentially more amenable to efficient parallelization. For instance, modularity reduces global memory references, thereby reducing memory-bus contention for shared-memory references and decreasing synchronization overhead. Second, modularity may facilitate *macro*-level performance improvements, even if there are increased *micro*-level performance penalties (such as additional subroutine-call overhead).

The latter point is exemplified by several empirical benchmarks performed using the *x*-kernel. These findings illustrate that efficient protocols may be created from modular, layered, and reusable software components if the process architecture minimizes context switching overhead and the message management scheme minimizes memory-to-memory copying (especially for large blocks of memory) [HP91]. For example, [HP91] demonstrated how the *x*-kernel's highly-layered OSTSA outperformed BSD's more monolithic OSTSA in terms of latency and throughput. Likewise, [OP90a] implemented the Sprite RPC protocol in the *x*-kernel using an efficient, highly-layered design that significantly outperformed the original monolithic Sprite implementation. One reason for the improvement was that the highly-layered *x*-kernel implementation allowed incoming and outgoing messages to bypass unnecessary layers of protocol processing (*e.g.*, skipping the IP layer when messages are bound for hosts on a local subnet).

[28] For instance, TCP uses the IP pseudo-header for checksum calculations.

3.3.2 The Flexibility and Extensibility Dimension

As described in Section 2.3, the OS Protocol Architecture (OSPA) manages software components that implement protocols, sessions, and messages. The OSPA design generally determines the overall flexibility and extensibility of an OSTSA.[29] The *flexibility* of an OSPA is characterized by how easily *existing* components may be recombined to form new configurations, and *extensibility* is characterized by how easily *new* components and services may be added to the OSPA. As discussed below, OSPA flexibility and extensibility dimensions involve support for *multiple protocol families*, various *OSPA component composition mechanisms* (*e.g.*, component configuration time, component composition order, and whether typed or untyped component composition is supported), and *OSPA component locations* within the OSTSA.

(1) Support for Multiple Protocol Families: 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.*, checksum computation algorithms). Many different protocol families exist, such as SNA, TCP/IP, XNS, and OSI.

Support for multiple protocol families is becoming increasingly important for both interoperability and performance reasons. Obviously, an OSPA that supports TCP/IP, OSI, XNS, and SNA will be able to communicate with far more host machines than an OSPA that only supports one protocol family. Moreover, different protocol families offer different types of services that favor certain applications while compromising performance for others [WM87]. In any layered protocol family it is difficult to achieve good performance at layer N without efficient support from layers below N. Therefore, application and network performance may decrease when protocol families are not designed to meet their specific requirements [OP90b]. For example, the TCP/IP protocol family does not specify a low-latency RPC service, which makes it difficult to efficiently support request/response-style applications (such as implementing a distributed file server on a LAN containing diskless workstations).

Some OSPAs support only a limited number of protocol families. For instance the V-kernel [Che88] only supports VMTP, Xinu [Com91b] only supports TCP/IP, and early versions of Amoeba [TRS+90, KvRvST91] only supported its high-performance RPC protocol graph. If only one protocol family is supported, then various special-purpose optimizations (*e.g.*, coding the protocols in assembly language [RST89]) may be used to improve performance. On the other hand, more flexible and extensible OSPA designs are required to support multiple protocol families. For example, System V STREAMS, BSD UNIX, the *x*-kernel, and the Conduit framework all provide general-purpose OSPAs that support multiple protocol families.

(2) OSPA Component Configuration Time: OSPA components (*e.g.*, representing protocols, sessions, and messages.) may be configured and/or reconfigured either statically or dynamically. Static configuration takes place at operating system boot time. In this case, the OSPA component configuration is "hard-coded," and available communication services are based only upon pre-ordained alternatives. Both BSD UNIX[30] and Xinu support static OSTSA composition. For example, user application programs that use the Internet protocol family on BSD UNIX may only select between the TCP and UDP transport protocols.

Dynamically configured OSPAs, on the other hand, compose some or all of their components while the system is running. A common method for supporting dynamic composition is to provide user applications with OSNAPI control operations that modify the OSPA protocol graph. For example,

[29] Since an OSKA typically supports other OS subsystems besides network computing, they are usually more difficult to change than the OSPA.

[30] This paper focuses on 4.3 BSD UNIX. 4.4 BSD incorporates a more flexible OSPA composition mechanism similar to System V UNIX STREAMS. However, the 4.4 BSD design is still in flux.

System V STREAMS allows user applications to link and/or unlink protocol components (called STREAM modules) via the `ioctl` system call. In addition, support for the dynamic linking and loading of executable code is useful for increasing extensibility.

The main advantage of statically configured OSPAs is that they may run very efficiently, since they are able to make assumptions about component ordering. For example, the BSD UNIX OSPA tightly couples the TCP and IP layers, which enables it to place header fields (such as the source and destination IP network addresses) at fixed-offsets in an *mbuf* message. On the other hand, dynamically configured OSPAs require more complex mechanisms to process messages whose header fields may contain a variable number of addresses, stored at variable offsets in the message headers.[31]

The main disadvantage of statically configured OSPAs is that they are inflexible. For instance, after any modifications, OSPA code must be recompiled, relinked, and restarted; carrying out these activities may require system downtime. Moreover, adding complicated extensions often requires changes to the design of OSPA software components and application programs. For example, when OSI and XNS support was added to the BSD UNIX kernel, many modifications were required to kernel- and user-level source code and system call interfaces [OTW85].

Dynamic configurations have several advantages over static configurations. First, they enhance flexibility and extensibility by enabling "dynamically tailoring" of OSPA components that selectively adapt to user application requirements and particular network environments. OSPA configurations may be specified by applications or they may be based upon dynamically changing feedback on network congestion, CPU load, and resource availability of networks and hosts [Sti92, Tsc91]. In addition, the ability to modify an OSPA at run-time may be important for systems that must be "highly available" (*e.g.*, systems such as an airline reservation system or telephone switching systems that cannot tolerate downtime).

(3) OSPA Component Composition Order: OSPAs that support dynamic configuration must provide some form of user interface to enable inserting and/or removing components. Two general approaches are to either allow components to be added or subtracted in an arbitrary order, or to enforce some constraints on the order such as "last-in, first-out" (LIFO). The *x*-kernel and Conduit framework provide OSPAs that support arbitrary component composition orders. System V STREAMS, on the other hand, only supports LIFO composition orders.

(4) Typed versus Untyped Component Composition: OSPAs supporting dynamic configuration may also provide either *typed* or *untyped* component composition. Typed composition is used to ensure that components are composed together in meaningful ways. This prevents, for instance, a TCP protocol component from being accidentally attached directly to an Ethernet driver, instead of an IP component. The Conduit framework is an example of a system that provides typed composition. Conversely, most OSPAs do not have standard facilities to ensure that arbitrary component combinations are semantically valid.

(5) OSPA Component Location: OSPA components may be implemented inside the OS kernel, in user-space, in off-board processors, or in some hybrid combination of these. Component location affects flexibility and extensibility, since it is generally harder to debug, develop, port, modify, and maintain OS kernel code, compared with code written in user-space [Cla82]. For example, when the OS runs in *kernel-mode* there is generally no protection against run-time errors. Therefore, erroneous kernel code may cause the entire OS to crash, whereas erroneous code running in user-mode only causes the associated user program to crash.

[31] Variable-offset headers may be necessary because the particular set of higher-layer protocol addresses is not known until a message arrives at the bottom of a protocol graph [OP91].

Process Architecture	(1) Coroutines, (2) horizontal (process-per-module)
Event Management	(1) absolute, (2) heap, (3) function call
Virtual Memory Remapping	none
Message Buffering	list-based
Multiplexing/Demultiplexing	(1) asynchronous, (2) layered, (3) ND, (4) ND
Flow Control	per-queue
Modularity	(1) uniform, (2) low coupling
Flexibility and Extensibility	(1) multiple, (2) dynamic, (3) LIFO, (4) untyped, (5) kernel-space

Table 2: STREAMS Profile

Component location also affects performance. Implementing components in user-space may result in poor performance due to the "boundary-crossing penalty" [OAHP90] that occurs when processes move between user- and kernel-mode (*e.g.*, as the result of a system call). The boundary-crossing penalty results from the overhead of demultiplexing, system calls, and context switching operations. For example, it requires at least 2 context switches and 3 system calls per-received-message to perform demultiplexing in user-space [MRA87]. On the other hand, performance overhead is greatly reduced if most OSPA components run in kernel-mode, since kernel data structures may be accessed directly (thereby reducing context switch overhead).

Two mechanisms used to reduce the overhead from crossing the user/kernel boundary are *upcalls* and *packet filters*. Upcalls are synchronous communication mechanisms that transfer control upwards from server to client [Atk88, Cla85]. The *x*-kernel uses upcalls as an optimization technique to reduce context switching overhead for incoming messages by allowing kernel processes to transform into user processes. Packet filters are a kernel resident, protocol independent packet demultiplexer that is used to reduce context switching overhead [MRA87]. A packet filter enables a user process to specify to the kernel which packet types it wants to receive. The kernel then performs the packet processing operations on behalf of the user process.

4 Survey of Existing OS Transport System Architectures

This section surveys the OS Transport System Architectures (OSTSA) for the System V UNIX, BSD UNIX, *x*-kernel, Choices, and Xinu operating systems. 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 gives an overview of some significant features (*e.g.*, the software components and process architecture) for each surveyed OSTSA. In addition, an OSTSA *profile* corresponding to the taxonomy illustrated in Table 1 is presented with each overview. These profiles were derived from articles in the open literature and from examining the source code, when available.

4.1.1 System V STREAMS

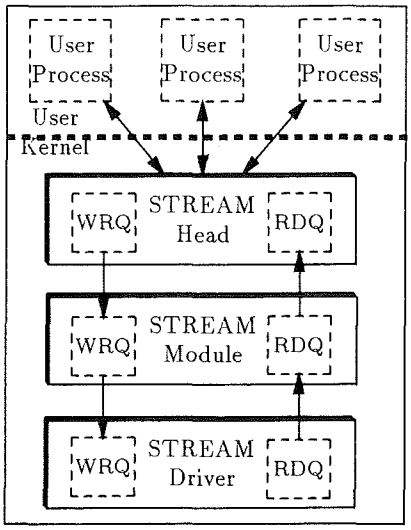

Figure 11: An Example Stream in System V STREAMS

The System V STREAMS model was originally developed to enhance the portability, reusability, and extensibility of the 8^{th} Edition Research UNIX serial-line I/O subsystem [Rit84]. The STREAMS design was initially oriented towards terminal drivers. It was later extended to support network protocols and local IPC, via *multiplexor drivers*, *STREAM pipes*, and *named FIFOS*. The STREAMS architecture emphasizes modular components that possess standard interfaces. Many other experimental OSTSAs (such as the *x*-kernel and the Conduit framework) are heavily influenced by the STREAMS architecture. Table 2 illustrates the OSTSA profile for System V UNIX STREAMS.

As shown in Figure 11, the three main layers in the System V STREAMS architecture include: the *STREAM head*, the *STREAM module*, and the *STREAM driver* layer [McG88]. Uniform service interfaces exist between each layer. A Stream[32] is a full-duplex "protocol processing and data transfer" path between a STREAM head and a STREAM driver. STREAM modules may be inserted and/or removed dynamically between the STREAM head and the STREAM driver. These modules implement protocol processing services like encryption, compression, reliable message delivery, and routing. The following bullets describe the System V STREAMS components in greater detail.

• **STREAM Heads:** The STREAM head is a special type of STREAM module. It is situated on "top" of a Stream, nearest to the user process. A STREAM head provides a queueing point where data and control information is exchanged between a distributed application (running as user processes) and a Stream (running in the kernel). The STREAM head is responsible for segmenting the user data (which may be produced as a continuous bytestream) into discrete messages. These messages flow "downstream" from the STREAM head, though zero or more STREAM modules, to the STREAM driver, where they are transmitted by a network controller to the appropriate remote host machines. Conversely, the driver also receives incoming messages. These messages then flow "upstream" through the modules to the STREAM head, where a user process may wait to retrieve them. The STREAM head also performs memory-to-memory copying to move data between a user process and kernel (*i.e.*, the System V kernel does not use virtual memory remapping techniques in its OSNAPI). Since a STREAM head runs in context of a user process, it may sleep when it is

[32]The uppercase word "STREAMS" refers to the overall System V OSTSA mechanism, whereas the word "Stream" refers to a particular path between a user application and a device driver.

blocked.[33]

• **STREAM Modules and Queues:** STREAM modules are analogous to "filter" programs in a UNIX shell pipeline. Data flows from a Stream head, through a stack of STREAM modules, to a STREAM driver. Each STREAM module performs its protocol processing operations on data it receives before sending the data along to the next module. Unlike a UNIX pipeline, however, data is passed as discrete messages between modules, rather than as a bytestream.

User processes may dynamically "push" and/or "pop" STREAM modules from a Stream (however, modules may only be inserted or removed in a "last-in, first-out" (LIFO) order). Each module contains a pair of *queues*, which are always allocated in read/write pairs (*i.e.*, every STREAM module, STREAM driver and STREAM head contains a queue pair). Queues are used for several purposes. First, they link STREAM modules together with other STREAM modules (a STREAM driver is linked below the "bottom" module of the Stream; likewise, a STREAM head is linked above the "top" module of the Stream). Second, they hold lists of messages sorted in priority order (messages may be ranked with up to 256 different priority levels). Finally, they contain pointers to a set of subroutines that implement the module's processing operations (*e.g.*, encrypting and decrypting data) and regulate layer-to-layer flow between modules.

Two important subroutines in this set are called "`put`" and "`service`." A `put` subroutine runs in response to synchronous or asynchronous events (*e.g.*, a user process sending a message downstream or a message arriving on a network interface). It performs protocol processing operations (such as handling high-priority messages like TCP urgent data) that must be invoked immediately. The `service` subroutine, on the other hand, is used for protocol processing operations that either do not execute in a short, fixed amount of time (*e.g.*, performing a three-way handshake to establish a peer-to-peer network connection) or that will block (*e.g.* due to layer-to-layer flow control[34]).

• **STREAM Drivers:** Like a STREAM head, a STREAM driver is also a special type of STREAM module.[35] There are two main categories of STREAM drivers: *device* drivers and *multiplexor* drivers. Device drivers exist at the "bottom" of a Stream. They typically manage hardware devices, and perform activities like handling network controller interrupts and converting raw packets into message data structures suitable for upstream modules.

A multiplexor driver may exist between a STREAM head and a STREAM driver, just like a STREAM module. Unlike a STREAM module, however, a multiplexor driver enables *multiple* Streams to link to it from "above" or "below." Multiplexor drivers 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). However, the STREAMS mechanism has no built-in support for flow control among multiplexor drivers. Therefore, STREAM multiplexor drivers require Stream implementors to develop additional "protocol-specific" software that performs message multiplexing and demultiplexing and flow control.

• **Messages:** Data is passed between STREAM heads, STREAM modules, and STREAM drivers in discrete chunks, using a standard abstract data type (ADT) called a *message*. A message is used to represent both data and control information (a message may have multiple data parts,

[33] STREAM heads block in several situations. One is when a user process performs a "blocking read" while waiting for messages to arrive on a network interface. Another occurs from "back-pressure" exerted by layer-to-layer flow control from "downstream" modules.

[34] In fact, the distinction between `put` and `service` subroutines was made to support flow control [Rit84]. Flow control occurs between the two nearest queues in a Stream that contain a `service` procedure.

[35] For example, unlike STREAM modules, STREAM heads and STREAM drivers cannot be "pushed" or "popped" onto a Stream dynamically.

but only *one* control part). Messages are passed upstream and downstream *by reference* to reduce memory-to-memory copying.

A message is represented by a <message control block, data control block, variable length data buffer> three-tuple. This three-tuple facilitates "logical" message duplication (that does not incur memory-to-memory copying costs) by sharing a single <data buffer> among <message control block, data control block> headers. The variable length data buffer has a default length of 64 bytes, though it may be allocated to be any power of two, up to a configuration-defined maximum limit.

• **Process Architecture:** System V STREAMS supports a variant of the horizontal process architecture. Conceptually, it provides a "process-per-module" architecture, in which one or more "logical" processes are associated with each STREAM module's `put` and `service` subroutines.[36] However, these subroutines run outside the context of any kernel or user process, and therefore, bypass the standard operating system kernel process scheduling mechanism. There are several reasons for this behavior. First, early versions of STREAMS did not support concurrent execution, since the System V kernel was single-threaded. Second, dedicating a standard kernel process for each STREAM module is highly consumptive of memory and CPU resources. For example, supporting a large number of modules, each with their own process state, would require many additional kernel stacks and process table slots, and would involve context switching overhead when moving messages between modules [Rit84].

The horizontal, "process-per-module" process architecture is emulated by scheduling and executing the `service` subroutines associated with the read/write queues in a STREAM module. `Service` subroutines interact in a *coroutine* manner. For example, when a queue's `service` subroutine is run, it performs protocol processing operations on all the messages waiting in its queue (note that due to multiplexor drivers, messages may have come from multiple upstream or downstream modules). By the time a `service` routine finishes its processing, it will have passed its processed messages to the appropriate STREAM components adjacent in the Stream. Any STREAM module that now has new messages in its read queue will have its `service` subroutine executed by a STREAMS scheduling mechanism. Note that `service` procedures are run only at certain times such as just before returning from a system call and just before a process is put to sleep.

One effect of this process architecture design is that the STREAM modules do not exist in the context of an OS process. Therefore, `put` and `service` subroutines cannot sleep if they must block (*e.g.*, due to flow control). If a subroutine detects that it must block, the currently executing `put` or `service` subroutine must *explicitly* save its state information before completing its current processing. In other words, the STREAMS mechanism has no provision for automatically retaining the state of blocked `put` or `service` routines.

4.1.2 BSD UNIX

BSD UNIX provides an OSTSA framework that supports multiple protocol families. This framework was designed originally to support the DARPA TCP/IP protocol family [LMKQ89]. Over time, other protocol families (*e.g.*, XNS, and OSI) have been incorporated into the framework [OTW85]. BSD supports the development of distributed applications that are independent of the underlying OSTSA

[36] Note that the OSF/1 UNIX implementation supports various granularity levels of STREAMS concurrency. From finest- to coarsest-grain, these concurrency levels are: (1) *queue-level* (*i.e.*, one light-weight process (LWP) for the STREAM module read queue, one LWP for the STREAM module write queue), (2) *queue-pair-level* (*i.e.*, one LWP shared by a STREAM module queue pair), (3) *module-level* (*i.e.*, one LWP shared across all instances of a STREAM module), and (4) *module-class-level* (*e.g.*, one LWP shared across a particular class of STREAM modules).

Process Architecture	(1) single-threaded, (2) vertical
Event Management	(1) relative, (2) linked list, (3) function call
Virtual Memory Integration	none
Message Buffering	list-based
Mulitplexing/Demultiplexing	(1) hybrid, (2) layered, (3) sequential-search, (4) single-item
Flow Control	ND
Modularity	(1) non-uniform, (2) high coupling
Flexibility and Extensibility	(1) multiple, (2) static, (3) static, (4) untyped, (5) kernel-space

Table 3: BSD UNIX Profile

protocols via a general-purpose OSNAPI called *sockets*.[37] Table 3 illustrates the OSTSA profile for BSD UNIX.

The concept of a *communication domain* is central to BSD's multiple protocol family design. Domains specify a set of related protocols and an address family. Protocols implement the standard domain socket types, *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 of fields, and order of fields) and a set of subroutines that interpret the address format (*e.g.*, to determine which subnet an IP message is intended for). BSD supports address families for the UNIX domain, Internet domain, XEROX NS domain, and the OSI domain.

There are three main layers in the BSD OSTSA design: the *socket layer*, *protocol layer*, and *network interface layer*. Like System V STREAMS, well-defined interfaces exist between each layer, although BSD generally places less emphasis on making the interfaces *uniform*.[38] A socket performs OSNAPI services that are similar to the System V STREAM head.[39] The protocol layer coordinates algorithms and data structures used to implement the protocol families that BSD supports. The network interface layer provides a well-defined software interface to network controllers. The following bullets describe the major BSD components in greater detail.

• **The Socket Layer:** A socket is a *typed* object that represents a bi-directional endpoint of communication. It serves as a queueing point for data that is transmitted and received between user applications (running as user processes) and the protocol layers (running in the kernel). A *socket descriptor* is used to identity an open socket. This descriptor is a small integer that indexes into a kernel data structure containing socket-related information (*e.g.*, send and receive buffer queues, the socket type, and the associated protocol layer control blocks). When a socket is created, this data structure 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 "naive" UNIX applications[40] to communicate transparently with different types of devices such as remote network connections, files, terminals, printers, and tape drives.

[37] Sockets augment the standard UNIX local IPC mechanisms: signals and pipes. Unlike pipes and signals, sockets allow arbitrary data communication between unrelated processes on local and remote host machines.

[38] Note that there is a difference between having "well-defined" interfaces and having uniform interfaces. The former simply means that data structures are accessed under the control of a subroutine, rather than accessed directly. The latter refers to having the same interface at multiple layers in a protocol graph.

[39] The primary difference between sockets and STREAM heads is that STREAM heads support up to 256 levels of message priority via the `getpmsg` and `putpmsg` system calls.

[40] Naive UNIX applications read from their *standard input* and write to their *standard output*.

• **The Protocol Layer:** BSD's protocol layer contains multiple protocol *sublayers* per protocol family. For instance, in the Internet protocol family, the TCP sublayer is connected over top of the IP sublayer. Each protocol sublayer maintains its own session state information. This session information is stored in *control blocks*, which are used to manage active end-to-end network connections. Control blocks that are used in the Internet domain include the `inpcb` (which stores a connection's host addresses and port number) and the `tcpcb` (which stores the TCP state machine variables such as sequence numbers, retransmission timer values, and statistics for network management). Each `inpcb` contains links to sibling `inpcbs` (which store session information for other active network connections in the protocol layer), links to the socket data structure associated with the protocol session, and other relevant information (*e.g.*, routing-table entries or network interface addresses). The session data structures that represent an active TCP connection consists of a <`socket, inpcb, tcpcb`> three-tuple.

• **The Network Interface Layer:** Messages arriving from the network are handled by interrupts rather than separate processes.[41] There are two levels of interrupts: SPLNET and SPLIMP. SPLNET has higher priority and is generated when a hardware device interrupts (*e.g.*, signaling that a message has arrived from a network controller). Hardware interrupts cannot be masked for very long without causing other OS devices to timeout and fail. Therefore, a second, lower priority software interrupt level named SPLIMP is used to invoke the higher-layer protocol processing. When an SPLNET hardware interrupt occurs, the incoming message is placed in the appropriate network interface protocol queue (*e.g.*, the queue associated with IP processing). Next, an SPLIMP software interrupt is posted, which informs 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 (a message might be discarded due to a checksum error) by a protocol, it typically ends up in a socket receive queue, waiting for a user process to retrieve it.

• **Mbufs:** BSD UNIX uses the mbuf data structure to manage messages as they flow between protocol layers. An mbuf's representation and its associated operations are similar to a System V STREAMS message. Mbuf management operations include subroutines for allocating and freeing mbufs and chains of mbufs and for adding and deleting data. These subroutines generally try to perform operations that minimize memory coping.

Mbufs are used for storing lists and chains of incoming messages and outgoing protocol segments, as well as other dynamically allocated data structures 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 via reference counting.

• **Process Architecture:** BSD uses a single-threaded, vertical process architecture residing entirely in the kernel. User processes enter the kernel by making a system call. Due to flow control, multiple user processes (that are sending data to "lower" protocol layers residing in the kernel) may simultaneously be blocked at the socket layer (and are therefore unable to continue processing messages down to the network interface layer). However, as incoming messages are passed up to "higher" protocol layers only one "process" is permitted to run.[42]

[41] Two reasons for using interrupts are (1) they avoid context switching overhead and (2) the BSD kernel is not multi-threaded.

[42] Note that when messages arrive from the network they are handled in the "bottom half" of the BSD kernel, which operates outside the context of a standard UNIX user-level process.

Process Architecture	(1) LWP, (2) vertical (process-per-msg)
Event Management	(1) relative, (2) linked list, (3) function call
Virtual Memory Remapping	complete
Message Buffering	graph-based
Multiplexing/Demultiplexing	(1) synchronous, (2) layered, (3) hashing, (4) single-item
Flow Control	per-process
Modularity	(1) uniform, (2) low coupling
Flexibility and Extensibility	(1) multiple, (2) dynamic, (3) arbitrary, (4) untyped, (5) kernel-space

Table 4: *x*-kernel Profile

4.1.3 x-kernel

The *x*-kernel is a modular, extensible OSTSA development environment designed to support OSPA and OSSA implementation and experimentation [HP91]. It was also designed to demonstrate that layering is not inherently detrimental to network protocol performance [OP90a]. The *x*-kernel supports *protocol graphs* (see Figure 3) that implement a wide range of standard and experimental protocol families, including TCP/IP, Sun RPC, Sprite RCP, VMTP, NFS, and Psync [PBS89]. Relationships between protocols are described via the protocol graph. Unlike BSD UNIX, whose OSPA is characterized by a static, relatively monolithic protocol graph, the *x*-kernel supports dynamic, highly-layered protocol graphs. Table 4 illustrates the OSTSA profile for the *x*-kernel.

The *x*-kernel's OSPA provides highly uniform interfaces to its services, which manage three fundamental software communication abstractions that commonly occur in network protocol graphs [HP91]: *protocol objects*, *session objects*, and *message objects*. These abstractions are supported by several reusable software components, including a *message manager* (an ADT used to encapsulate messages that are exchanged between session and protocol objects), a *map manager* (used for demultiplexing), and an *event manager* (used for timer-driven activities like TCP's adaptive retransmission algorithm). In addition, the *x*-kernel provides a library containing *micro-protocols*, which are highly modular software components that implement services common to many network protocols such as sliding window adaptive retransmission schemes, request/response RPC mechanisms, and "blast" algorithms with selective retransmission [OP91]. The following bullets describe the *x*-kernel's major components in greater detail.

- **Protocol Objects:** Protocol objects are software abstractions used to implement network protocols in the *x*-kernel. The *x*-kernel implements a *protocol graph* by combining one or more *protocol objects* in well-defined ways. A protocol object contains a standard set of subroutines that provide uniform interfaces for two major services: first, protocol objects create and destroy *session objects* (described in the next bullet below); second, protocol objects demultiplex message objects onto the appropriate higher-layer session objects (the *x*-kernel uses the *map manager* abstraction to implement efficient demultiplexing). The map manager associates external identifiers (*e.g.*, TCP port numbers) with internal data structures (*e.g.*, session control blocks). It is implemented by a chained-hashing scheme with a single-item cache.

- **Session Objects:** A session object maintains state information associated with an end-point of a network connection. For example, a session object may store the current state of an active TCP finite state machine. Multiple session objects may be associated with a given protocol object (the

Process Architecture	(1) LWP, (2) hybrid (process-per-buffer)
Event Management	ND
Virtual Memory Integration	none
Message Buffering	list-based
Mulitplexing/Demultiplexing	(1) ND, (2) layered, (3) ND, (4) ND
Flow Control	ND
Modularity	(1) uniform, (2) low coupling
Flexibility and Extensibility	(1) multiple, (2) dynamic, (3) arbitrary, (4) typed, (5) user-space

Table 5: Conduit Framework Profile

protocol object dynamically creates and disposes the session objects). Operations on session objects primarily involve "layer-to-layer" activities such as exchanging messages between higher-level and lower-level sessions. Note that the *x*-kernel OSPA framework only specifies layer-to-layer operations on session objects. In particular, it does not provide any standard support for peer-to-peer OSSA activities, such as connection management, error detection, etc.[43]

• **Message Objects:** Message objects are instances of the message manager ADT. Messages flow "upwards" or "downwards" through graphs of session and protocol objects. In order to decrease memory-to-memory copying and to efficiently implement message operations, message objects are represented with a graph-based data structure. This graph-based scheme uses "lazy-evaluation" that avoids unnecessary data copying [HMPT89]. It also stores message headers separately from the message data to reduce the cost of protocol encapsulation (*e.g.*, prepending or stripping headers).

• **Process Architecture:** The *x*-kernel employs a "process-per-message" vertical process architecture that resides in the OS kernel. A pool of light-weight processes is cached in the kernel. When a message arrives at a network interface, a separate light-weight process is dispatched from the pool to shepard it upwards through session objects associated with protocol layers in the protocol graph. In general, only one context switch is required to shepard a message throughout the protocol graph, regardless of the number of intervening protocol layers. The *x*-kernel also supports another optimization that reduces context switching overhead by allowing user processes to transform into kernel process during message output (via system calls) and kernel processes to transform into user processes during message input (via upcalls) [Cla85].

4.1.4 The Choices "Conduit framework"

The Conduit framework provides the OSPA, OSSA, and OSNAPI levels for the Choices operating system [CRJ87]. Choices is being developed to study how suitable object-oriented techniques are for the design and implementation of OS kernel and networking facilities.[44] For example, the design of ZOOT (the Choices TCP/IP implementation) uses object-oriented language constructs and design methods (*e.g.*, inheritance, dynamic binding, and delegation [ZJ91]) to implement the TCP state machine in a highly modular fashion. Together, Choices and the Conduit framework provide a general-purpose OSTSA. Table 5 illustrates the OSTSA profile for the Choices Conduit.

[43] The Avoca project builds upon the basic *x*-kernel facilities to provide these peer-to-peer services [BO91].

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

There are three major components in the Conduit framework: *Conduits*[45], *Conduit Messages*, and *Conduit Addresses*. The *Conduit* is a bi-directional communication abstraction, similar to a System V STREAM module. It exports operations that allow other Conduits to link together with it and to exchange messages with adjacently linked Conduits. *Conduit Messages* are the typed objects exchanged between Conduits. *Conduit Addresses* are utilized by Conduits to determine where to send a Conduit Message. All three components are described briefly in the following bullets.

- **Conduits:** A Conduit provides the basis for implementing an end-to-end network protocol such as TCP or TP4. It is represented as a C++ base class providing two sets of fundamental operations that may be redefined by subsequent subclasses. The first set of operations implement network protocol graphs by connecting and disconnecting Conduits. The second set of operations enable messages to be inserted into the "top" and "bottom" of a Conduit. A Conduit has two ends for processing data and control messages: the top end corresponds to messages flowing *down* from the application; the bottom end corresponds to messages flowing *up* from the network interface.

- **Conduit Subclasses:** The Conduit framework uses C++'s inheritance and dynamic binding mechanisms to represent the commonality between the Conduit base class and its various subclasses. These subclasses are *specializations* of abstract network protocol classes such as *Virtual Circuits* and *Datagrams*. Therefore, the Conduit framework defines two subclasses that provide additional services and interfaces: `Virtual_Circuit_Conduit` and `Datagram_Conduit`. Both subclasses export the connect, disconnect, and message insertion services inherited from the Conduit base class. However, they also extend their class interface by supplying operations that implement their additional services. For example, `Virtual_Circuit_Conduits` provide an interface for managing peer-to-peer "sliding window" flow control. They also specify 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 subclass specialization, resulting in `TCP_Conduits` and `Ethernet_Conduits`.

- **Conduit Messages:** All messages that flow between Conduits have a particular message type. The message type indicates the contents of a message (*e.g.*, its header and data format), and specifies the operations it may perform. Messages are represented by a C++ base class that provides a foundation for subsequent inherited subclasses. Different message subclasses are associated with different Conduit subclasses (that in turn represent different network protocols). For example, there are `IP_Message` and `TCP_Message` subclasses that correspond to the IP Conduits and TCP Conduits, respectively. Conduit messages subclasses may also encapsulate other messages. For instance, an IP message contains a TCP message in its data portion.

- **Conduit Addresses:** The Conduit framework uses addresses to determine where to send Conduit messages. The two main types of addresses are *explicit* and *implicit*. Explicit addresses are used for entities like Internet IP addresses or port numbers, which have a "well-known" format. Implicit addresses are used by connection-oriented protocols to identify session control blocks that are related to active network connections. For example, a TCP connection descriptor is identified by its "association," which consists of a <*local port*, *local address*, *remote port*, *remote address*> four-tuple.

- **Process Architecture:** The relationship of OSKA processes to Conduits and Conduit messages is not uniformly specified in the Conduit framework. Subsequent versions may use "walker-processes," which are similar to the *x*-kernel "process-per-message" mechanism. Each walker-process shepards one message up or down the protocol graph. Depending on flow control, a user process may be able to walk outgoing messages most of the way down the protocol graph. In this scheme, there are

[45] In the discussion below, the "Conduit framework" refers to the overall OSTSA, whereas "Conduit" corresponds to an abstract data type (ADT) used to construct and coordinate various network protocols.

Process Architecture	(1) HWP, (2) horizontal (process(es)-per-protocol)
Event Management	(1) relative, (2) linked list, (3) message passing
Virtual Memory Remapping	none
Message Buffering	list-based
Multiplexing/Demultiplexing	(1) asynchronous, (2) layered, (3) sequential-search, (4) none
Flow Control	per-queue
Modularity	(1) uniform, (2) low coupling
Flexibility and Extensibility	(1) multiple, (2) static, (3) static, (4) untyped, (5) kernel-space

Table 6: Xinu Profile

as many processes as there are flow control buffers in the chain of events between the application and network interface layer. In addition, one extra process is also required to transfer messages between Conduits that do not contain any buffers, and hence may not block due to flow control.

4.1.5 Xinu

The Xinu network computing software was developed as the OSPA for the Xinu OS [Com91b]. It is intended primarily as a pedagogical tool, emphasizing clarity of design and implementation over high-performance. Xinu currently supports the entire TCP/IP protocol family, including ARP, ICMP, IP, UDP, TCP, and SNMP [Com91a]. By default, Xinu is configured to support *gateway* operations, implementing multiple device drivers, Internet routing and network management. Table 6 illustrates the OSTSA profile for the Xinu operating system.

To enhance system clarity and to simplify the OSPA design, the Xinu TCP/IP process architecture uses one or more separate heavy-weight processes (HWP) to implement each TCP/IP protocol component that requires timer-driven processing. The main components in the Xinu system are ports, queues, messages, and processes (*e.g.*, for TCP input, output, and timer protocols, and the IP protocol). The following bullets describe these components in more detail.

• **Ports and Message-Passing:** Each major protocol process in Xinu's TCP/IP design executes in one or more separate HWP address spaces. Therefore, some form of synchronous or asynchronous mechanisms are provided for interprocess communication (IPC). The two main IPC mechanisms in Xinu are *asynchronous ports* and *synchronous message-passing.*

A Xinu port is a fixed-length rendezvous point. They are used in several places in the Xinu architecture (*e.g.*, to queue incoming segments between the IP process and the TCP process). Although ports exist independently of any process, processes use them to synchronize and communicate by en-queueing and de-queueing messages. Ports use semaphores to guarantee mutual exclusion. If the port is full, send operations will block a *producer* process (*e.g.*, a user process sending data to a TCP process). Likewise, if the port is empty, receive operations will block a *consumer* process (*e.g.*, a TCP input process that is awaiting the arrival of IP messages to process).

Xinu's message-passing scheme combines two mechanisms: (1) a synchronous notification mechanism that uses procedure calls to send a message directly from one process to another and (2) a separate queue that buffers variable-sized messages. Message-passing is also used for several purposes (*e.g.*, to deliver incoming messages from the network interface to the IP process).

There are two main distinction between ports and message-passing. First, ports provide a general-purpose queueing point for variable-sized messages, but message-passing only transmits a small 4-byte notification to a process (which might indicate that a separate queue now contains new messages). Second, ports allow asynchronous IPC, since a sending process can enqueue the message into the port and continue executing it protocol activities (as long as the port is not completely full). Message-passing, on the other hand, are synchronous, since processes performing a message-passing send or receive block until the other process completes the rendezvous.

• **Protocol Decomposition and Process Architecture:** Xinu's process architecture is tightly coupled to its protocol decomposition. Xinu employs multiple HWPs to implement the TCP/IP protocol family. The processes are scheduled by the Xinu OS scheduler. As with System V STREAMS modules, processes are scheduled to run when messages become available on ports and queues shared between two HWPs. The main processes are the *IP process*, the *TCP input and output* processes, and the *TCP timer process*, which are described in the following paragraphs.

The IP process reads and writes to multiple input and output queues (one for each network device interface).[46] It also has an input and output port used to store incoming and outgoing TCP segments. These queues and ports allow the various protocol processes to run concurrently.

The TCP input and output processes share access to TCP *Transmission Control Blocks* (TCBs). There is one TCB per active connection. TCBs serve the same purpose as the BSD `inpcb` and `tcpcb` data structures described in Section 4.1.2. the TCP input and output operations are performed by two separate processes that run concurrently. The TCP output process handles segmentation and data transmission. The TCP input process handles reassembly and demultiplexing.

The TCP timer process handles asynchronous event management for all the TCP/IP protocols. For example, it schedules retransmission timeouts by inserting an "event" (*i.e.*, the delay time, a message to be sent, and a port to send it to) into a *delta-list*. If the delay time expires before the event is canceled, the timer process sends the message to the specified port.

In addition to these TCP/IP processes, each user-level application is also associated with its own HWP. In the Xinu architecture, application processes are distinct from the kernel processes (unlike the *x*-kernel, where user-level processes may migrate into kernel space and vice versa). This approach leads to higher synchronization overhead, however, since multiple context switches are required to move messages from kernel-space to user-space.

• **Message Management:** Xinu's message management is similar to BSD UNIX's mbufs. It uses a hybrid scheme that support both large datagrams and linked lists of small buffers. Xinu pre-allocates many small buffers that hold a single message and several larger buffers for large messages. Physical data copies are generally only performed when reassembling large messages at the IP layer.

4.2 System Comparisons

This section compares and contrasts the five surveyed OSTSAs by the taxonomy dimensions and alternatives presented in Table 1.

4.2.1 Comparison of OS Kernel Architecture (OSKA) Dimensions

In this section we compare the five operating systems according to the OSKA dimensions described in Section 3.1.

[46] Since ICMP and UDP are relatively simple protocols, they run in the context of the IP process.

The Process Architecture Dimension: The surveyed OSTSAs cover a range of process architectures, although none of the surveyed OSTSAs provide comprehensive multi-processor support.[47] BSD UNIX and System V STREAMS have a single-threaded OSKA. Concurrent programming abstractions are provided by light-weight process facilities in the Conduit framework and the *x*-kernel, and by heavy-weight processes in Xinu.

System V STREAMS and Xinu use variants of the horizontal process architecture. System V STREAMS uses a "virtual process-per-module" approach.[48] Xinu's implementation uses a "heavy-weight process(es)-per-protocol" scheme, where each TCP/IP protocol component runs in one or more heavy-weight processes.

The *x*-kernel and BSD UNIX use variants of the vertical process architecture. The *x*-kernel uses a pure "process-per-message" approach that supports highly-layered protocol graphs without incurring excessive context switching and IPC overhead. BSD UNIX uses a vertical approach that behaves differently depending on whether messages are flowing "up" or "down" through a protocol graph. BSD allows multiple processes into the kernel for outgoing messages, but only one process for incoming messages.

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

The Event Management Dimension: BSD UNIX and *x*-kernel store pointers to subroutines in callout lists. This allow arbitrary subroutines to be called when a timer expires. System V maintains a heap-based callout table, rather than a sorted list or array. Xinu stores control messages on a *delta-list*, and passes a message to the appropriate pre-registered port if the timer associated with event expires.

The Virtual Memory Remapping Dimension: Recent versions of *x*-kernel provide virtual memory remapping [OAHP90]. Xinu, the Conduit framework, System V STREAMS and BSD UNIX, on the other hand, do not provide this support.

4.2.2 Comparison of OS Protocol Architecture (OSPA) Dimensions

In this section we compare the five operating systems according to the OSPA dimensions described in Section 3.2.

The *x*-kernel generally specifies the service interfaces for all its OSPA components more comprehensively than the other surveyed OSTSAs. For example, it provides uniform interfaces for operations that manage protocol, session, and message objects. In addition, it also specifies uniform interfaces and provides implementations for event management, and multiplexing and demultiplexing. System V STREAMS, on the other hand, only specifies various types of STREAM module interfaces and components. For example, every STREAM module contains a read/write queue pair and also uses certain standard subroutines to regulate layer-to-layer flow. The Conduit framework does not stipulate a Conduit's internal structure at all. In particular, sessions, multiplexing, and demultiplexing are not systematically specified by the Conduit framework.

The Message Management Dimension: System V STREAMS messages and BSD mbufs use a linear-list-based approach. The *x*-kernel, on the other hand, uses a graph-based approach. Since the *x*-kernel is designed to support highly-layered protocol graphs, it requires the more complex graph-based

[47] Although some commercial versions of STREAMS (*e.g.*, OSF/1) provide multi-threaded, light-weight process implementations.

[48] As described in Section 4.1.1, the standard System V STREAMS approach is "virtual" because it does not allocate a "real" OS process per module.

buffering scheme to efficiently handle the additional encapsulation and minimize memory-to-memory copying between layers.

The Multiplexing and Demultiplexing Dimension: The five OSTSAs possess a wide range of multiplexing and demultiplexing strategies. The *x*-kernel provides the most comprehensive support for these operations. It provides an efficient hashing-based mechanism, with a single-item cached that is optimized for different address sizes in a highly-layered protocol graph. The other systems provide less systematic mechanisms.

For example, the Conduit framework and System V STREAMS leave the design and implementation of multiplexing and demultiplexing to implementors of its Conduit subclasses. However, for outgoing messages, the Conduit framework involves an extra multiplexing operation compared to the *x*-kernel scheme. In *x*-kernel, outgoing messages simply make a subroutine call to transfer messages flowing downward from session object to session object. A Conduit, on the other hand, searches for the session object connection descriptor associated with the lower-level conduit. Xinu moves messages between protocol components using either ports or message-passing.

BSD UNIX's multiplexing and demultiplexing mechanisms differ according to protocol sublayer and protocol family. For instance, the IP layer 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, BSD's 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 for applications that do not form message-trains [MD91].

The Flow Control Dimension: Most OSTSAs do not provide uniform flow control mechanisms. System V STREAMS is an exception. Flow control between modules is voluntary, though each STREAM module contains high and low "watermarks" to manage flow control between its adjacent neighbors. Downstream flow control operates from the "bottom up." If all STREAM modules on a Stream cooperate, it is possible to apply "back-pressure" all the way up a stack of STREAM modules to the user process. For example, if the network is too congested to accept new messages (or if messages are being sent by a process faster than they can be 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 process continues all the way up to the STREAM head. "Back-enabling" (*i.e.*, causing previously block `service` subroutines to execute) is used to unblock flow controlled queues when congestion subsides.

In BSD UNIX, flow control occurs at several places throughout the OSPA. At the socket level, flow control is voluntary, using the "high and low watermark fields" stored in the socket data structure. If the act of performing a `send` operation would exceed a socket's highwater mark, the BSD kernel puts the sending process to sleep. Unlike System V, BSD UNIX has no standard mechanism for apply back-pressure between the OSPA protocol sublayers; it simply discards messages when its buffers become overwhelmed. At the network interface layer, queues are used to buffer messages between the network controllers and the lowest-level protocol (*e.g.*, IP). The queues have a maximum length that serves as a simple form of flow control. For example, subsequent incoming messages are dropped if these queues become full.

In Xinu, flow control is provided by ports that use semaphores to synchronize senders and receivers (*e.g.*, writing to a full port blocks a process, as does reading from an empty port). For operations that cannot block (such as network interface subroutines that run at hardware interrupt levels) a subroutine tests whether a send or receive will block. This allows messages to be discarded if they cannot be handled promptly.

The *x*-kernel and the Conduit framework provide less-uniform flow control support. The *x*-kernel supplies very coarse-grained flow control by discarding a message if there are no light-weight processes

available to shepard it up the protocol graph. The Conduit framework does not provide a standard mechanism to manage flow control between modules in a given stack of Conduits. Each Conduit hands 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 inserter may either dump the message or save it for later). This approach allows particular Conduit to decide whether it is a "message-dumping" entity or a "patiently-blocking" entity.

4.2.3 Comparison of Software Quality Dimensions

The OSPA for System V, the *x*-kernel, and Choices operating systems were are all explicitly developed to simplify and improve network protocol software development and maintenance. In particular, the Conduit framework uses object-oriented techniques like inheritance, dynamic binding, and delegation to increase protocol component reuse. In the following, we discuss dimensions described in Section 3.3.

The Modularity Dimension: The OSPAs for System V STREAMS, the *x*-kernel, and the Conduit framework are all highly modular. The STREAMS design emphasizes uniform interfaces between processing components in a Stream. For example, the same `put` and `service` interface is used to pass messages between a STREAM head, STREAM modules, and a STREAM driver.

The *x*-kernel emphasizes a single uniform protocol interface for its protocol, session, and message objects. In particular, its protocol objects dictate a "connection-setup" paradigm for all the supported network protocols (even connectionless protocols like UDP and IP) to achieve uniformity. Note that enforcing this degree of uniformity penalizes connectionless protocols to some extent, since sending a connectionless datagram in the *x*-kernel requires explicitly opening and closing a session in each protocol graph layer.

The Conduit framework avoids this "single uniform interface" penalty by providing several uniform interfaces, which are related by inheritance to the Conduit base class. Each interface differs in accordance to the abstract "class" of protocol involved. For instance, `Virtual_Circuit_ Conduits` have a different interface from the `Datagram_Conduits`, although they are related to the common Conduit base class.

Xinu uses uniform interfaces to reduce design and implementation complexity. For example, the local host interface and network interfaces use the same IP queueing structures. However, the Xinu system's modularization is tightly coupled to the TCP/IP architecture. For example, the subroutines implementing TCP correspond to states in the TCP finite state machine (*e.g.*, `tcpestablished`, `tcpclosewait`, `tcplisten`, etc.), making it difficult to directly reuse these interfaces for other protocol suites.

BSD UNIX does not emphasize uniform interfaces throughout the OSTSA. Its OSPA has uniform interfaces only for certain subroutines that form the boundary between the socket layer and the protocol layer. However, the system is not designed to facilitate reuse *between* communication domains (*e.g.*, each domain has incompatible *address family* formats [LMKQ89]). Although protocol sublayers have well-defined interfaces, these interfaces are not uniform between the layers. For example, a different interface is used for passing messages between the TCP and IP protocols than is used to pass messages between the IP protocol and network interfaces.

The Flexibility and Extensibility Dimension: All the surveyed OSPAs support multiple protocol families, with the exception of Xinu (which only supports the TCP/IP protocol family). In addition, System V STREAMS, the *x*-kernel, and the Conduit framework all support dynamic protocol graph reconfigurations. In System V, user processes manipulate Stream configurations dynamically by "pushing" and "popping" STREAM modules [UNI90]. STREAM modules can only be composed in a "last-in, first-out" (LIFO) order, however. The Conduit framework and the *x*-kernel, on the

other hand, do not impose this restriction. They both allow *arbitrary* OSPA configurations, which are supported by automated tools. The *x*-kernel provides a graphical user interface for specifying OSPA component ordering. It also allows higher-layer protocols to dynamically open lower-layer session objects. This allows a form of "late binding," since the particular characteristics of session object returned at run-time can vary [HP91]. The Conduit framework provides a name server that manages registered Conduits and enables Conduits to be attached together dynamically to form new communication abstractions [Zwe91].

Neither BSD UNIX nor Xinu provide any means to dynamically reconfigure the protocol graph, so protocol ordering relationships are completely established at system boot-time. It is not possible to reconfigure the OSPA while the system is running.

The Conduit framework is the only OSTSA that supports *typed* OSPA composition. Typed composition ensures that Conduits are joined dynamically in a meaningful manner. It uses a handshake scheme called "double-dispatch" when linking two Conduit together. Double-dispatching is a mechanism that allows two Conduits to determine each other's type. It is used to check that both Conduits belong to the same "family" of Conduit subclasses. On the other hand, neither the *x*-kernel nor System V STREAMS provide type-checked component composition.

Finally, with the exception of the Conduit framework, all OSPAs are implemented in the kernel, in order to improve performance. The Conduit framework runs in user-space, to improve its flexibility and extensibility by end-users.

5 Summary

Distributed application performance is influenced by multiple factors that interact in complicated ways. These factors involve the communication infrastructure as well as the computing infrastructure. Due to improvements in the communication infrastructure, the computing infrastructure is now the bottleneck for distributed applications running on high-speed networks. In particular, end-to-end user application performance depends heavily on how efficiently the computing infrastructure moves messages through protocol graphs.

This paper describes a virtual machine model (known as the OS Transport System Architecture (OSTSA)) that illustrates the levels of abstraction and services in the computing infrastructure. A taxonomy of the OSTSA levels is created and used to compare and contrast different alternatives that are found in five existing commercial and experimental operating systems. An important observation from these comparisons are that most existing operating systems lack explicit, efficient support for real-time operations and parallel protocol processing.

Our research group at University of California, Irvine is currently using this taxonomy to organize our research on OSTSA designs that are both highly modularity and efficient [BSS92]. We are developing an environment for analyzing and experimenting with various strategies for incorporating network protocols over top of high-speed communication links. In order to support the development of protocol designs that precisely meet multimedia application requirements and underlying network characteristics, we are developing a system called ADAPTIVE, which stands for "A Dynamically Assembled Protocol Transformation, Integration, and Validation Environment." Using ADAPTIVE, we are attempting to match diverse multimedia applications to a wide range of network characteristics.

Based upon our survey of the literature and existing systems, we view the following as important open research issues:

- Which OSTSA levels most impact on distributed system performance? Furthermore, how should OSTSAs be structured in order to increase their flexibility, extensibility, and perfor-

mance? For example, which choices from among the taxonomy dimensions and alternatives most improve overall communication performance?

- Which process architecture and parallelism models result in the highest performance, and under what conditions (*e.g.*, application requirements and network characteristics) are certain models preferred?
- Which OSTSA profiles are best suited for multimedia applications running in high-speed 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?

References

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

[AH91] David Anderson and George Homsy. A Continuous Media I/O Server and Its Synchronization Mechanism. *IEEE Computer*, pages 51–57, October 1991.

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

[BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall International Series in Computer Science, 1990.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, Englewood Cliffs, NJ, 1986.

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

[Bla91] U. Black. *OSI: A Model for Computer Communications Standards.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[BO91] Don Batory and Sean W. O'Malley. The Design and Implementation of Hierarchical Software Systems Using Reusable Components. Technical Report TR 91-22, Department of Computer Sciences, University of Texas at Austin, Austin, Texas., June 1991.

[Bro88] Laurence M. Brown. Networking Architecture and Protocol. In *UNIX System Software Readings*, pages 81–107. AT&T Unix Pacific Co. Ltd, 1988.

[BSS92] Donald F. Box, Douglas C. Schmidt, and Tatsuya Suda. Alternative Approaches to ATM/Internet Interoperation. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, pages 1–5. IEEE, 1992.

[CG91] David R. Cheriton and Hendrik A. Goosen. Paradigm: A highly scalable shared-memory multicomputer architecture. *IEEE Computer*, 24(2):33–46, February 1991.

[Che86] David R. Cheriton. VMTP: A Transport Protocol for the Next Generation of Communication Systems. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 406–415, Stowe, VT, August 1986. ACM.

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

[Che88] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3), March 1988.

[Che89] Greg Chesson. XTP/PE Design Considerations. In *Proceedings of the 1st International Workshop on High-Speed Networks*, May 1989.

[CJRS89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[Cla82] David D. Clark. Modularity and Efficiency in Protocol Implementation. *Network Information Center RFC 817*, pages 1–26, July 1982.

[Cla85] David D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, Shark Is., WA, 1985.

[Com91a] Douglas E. Comer. *Internetworking with TCP/IP Vol I: Principles Protocols, and Architecture, 2nd edition.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Com91b] Douglas E. Comer. *Internetworking with TCP/IP Vol II: Design, Implementation, and Internals.* Prentice Hall, Englewood Cliffs, NJ, 1991.

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

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

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

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

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

[GA91] Ramesh Govindan and David P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 68–80, October 1991.

[GDFR90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an application program. In *Usenix 1990 Summer Conference*, pages 87–95, June 1990.

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

[GL89] Karen D. Gordon and Cathy J. Linn. Strategic Defense System Distributed Operating System R&D Review and Recommendations. Technical Report IDA Paper P-2142, Institute for Defense Analyses, April 1989.

[Gos91] A. Goscinski. *Distributed Operating Systems: the Logical Design.* Addison-Wesley, Reading, Mass, 1991.

[Gre91] Paul E. Green. The Future of Fiber-Optic Computer Networks. *IEEE Computer*, pages 78–87, September 1991.

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

[HB85] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, SE-11(8):749–757, 1985.

[HD89] Christian Huitema and Assem Doghri. A High Speed Approach for the OSI Presentation Protocol. In H. Rudin and Robin Williamson, editors, *Protocols for High Speed Networks*. IFIP, North-Holland, 1989.

[HEHK92] Bernd Hofmann, Wolfgang Effelsberg, Thomas Held, and Hartmut Konig. On the Parallel Implementation of OSI Protocols. In *Proceedings of the IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, February 1992.

[Hen80] Kathryn L. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and their Application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[HK81a] S. Henry and D. Kafura. Software quality metrics based on interconnectivity. *Journal of Systems and Software*, 2(2):121–131, 1981.

[HK81b] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineerings*, 1981.

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

[Hol91] Gerald J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, NJ, 1991.

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

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

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

[KvRvST91] M. F. Kaashoek, Robbert van Renesse, Hans van Staveren, and A. S. Tanenbaum. FLIP: an Internetwork Protocol for Supporting Distributed Systems. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, July 1991.

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

[McG88] Gilbert J. McGrath. Streams Technology. In *UNIX System Software Readings*, pages 49–79. AT&T Unix Pacific Co. Ltd, 1988.

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

[Mey89] Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, 1989.

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

[MRA87] Jeffrey C. Mogul, Richard F. Rashid, and Michal J. Accetta. The Packet Filter: an Efficient Mechanism for User-level Network Code. In *The Proceedings of the 11th Symposium on Operating System Principles*, November 1987.

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

[Mul90] Sape J. Mullender. *Distributed Systems.* ACM Press, 1990.

[Mye78] Glenford. J. Myers. *Composite/Structured Design.* Van Nostrand Reinhold, 1978.

[NRS90] A. N. Netravali, W. D. Roome, and K. Sabnani. Design and Implementation of a High Speed Transport Protocol. *IEEE Transactions on Communications*, 1990.

[OAHP90] Sean W. O'Malley, Mark B. Abbott, Norman C. Hutchinson, and Larry L. Peterson. A Transparent Blast Facility. *Journal of Internetworking*, 1(2), December 1990.

[OP90a] Sean W. O'Malley and Larry L. Peterson. A Highly Layered Architecture for High-Speed Networks. In *Proceedings of the 2nd International Workshop on High-Speed Networks*, November 1990.

[OP90b] Sean W. O'Malley and Larry L. Peterson. A New Methodology for Designing Network Software. Technical Report TR 90-29, Department of Computer Science, University of Arizona, Tucson, Ariz., August 1990.

[OP91] Sean W. O'Malley and Larry L. Peterson. A Dynamic Network Architecture. Technical report, Department of Computer Science, University of Arizona, Tucson, Ariz., October 1991.

[OTW85] J. O'Toole, C. Torek, and M. Weiser. Implementing the XNS Protocol for 4.2 BSD. In *Proceedings of the 1985 Winter USENIX Conference*, pages 90–97, 1985.

[Par72] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), December 1972.

[Par79] David L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, March 1979.

[Par90] Gurudatta Parulkar. The Next Generation of Internetworking. *ACM Computer Communication Review*, 20(1):18–43, January 1990.

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

[PC91] Gee-Swee Poo and Boon-Ping Chai. Modularity Versus Efficiency in OSI System Implementations. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, pages 950–959, Bal Harbour, FL, April 1991. IEEE.

[PCW83] David L. Parnas, P.C. Clements, and D.M. Weiss. Enhancing Reusability with Information Hiding. *ITT Proceeding of the Workshop on Reusability in Programming*, 1983.

[PPA+90] Joseph C. Pasquale, George C. Polyzos, Eric W. Anderson, Kevin R. Fall, Jonathan S. Kay, Vachaspathi P. Kompella, Scott R. McMullan, and Dipti Ranganathan. Network and Operating System Support for Multimedia Applications. Technical Report ??, University of California, San Diego, 1990.

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

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

[RST89] R. Van Renesse, H. Van Staveren, and A. S. Tanenbaum. Performance of the Amoeba Distributed Operating System. *Software – Practice and Experience*, 19:223–234, March 1989.

[SB91] Richard W. Selby and V. R. Basili. Analyzing Error-Prone System Coupling and Cohesion. *IEEE Transactions on Software Engineering*, 17(2):141–152, February 1991.

[Sel88] Richard W. Selby. Generating Hierarchical System Descriptions for Software Error Localization. In Lee J. White, editor, *Second Workshop on Software Testing, Verification, and Analysis*, pages 89–97. IEEE Computer Society, 1988.

[Sha91] A. Udaya Shankar. Modular Design Principles for Protocols with an Application to the Transport Layer. *Proceedings of the IEEE*, pages 1687–1707, December 1991.

[Ste92] Peter Steenkiste. Analysis of the Nectar Communication Processor. In *Proceedings of the IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, February 1992.

[Sti92] Burkhard Stiller. PROCOM: A Manager for an Efficient Transport System. In *Proceedings of the IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, February 1992.

[Sun90] Sun Microsystems. *Transport Level Interface Programming*, April 1990.

[Tan88] Andrew S. Tanenbaum. *Computer Networks (Second Edition)*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Tan90] Andrew S. Tanenbaum. *Structured Computer Organization (Third Edition)*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Englewood Cliffs, NJ, 1992.

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

[TR85] Andrew S. Tanenbaum and Robbert Van Renesse. Distributed Operating Systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[TRG+87] Avadis Tevanian, Richard Rashid, David Golub, David Black, Eric Cooper, and Michael Young. Mach Threads and the Unix Kernel: The Battel for Control. Technical Report CMS-CS-87-149, Carnegie Mellon University, August 1987.

[TRS+90] Andrew S. Tanenbaum, Robbert Van Renesse, Hans Van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido Van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–63, December 1990.

[Tsc91] Christian Tschudin. Flexible Protocol Stacks. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 197–205, Zurich Switzerland, September 1991. ACM.

[UNI90] UNIX Software Operations. *UNIX System V Release 4 Programmer's Guide: STREAMS.* Prentice Hall, 1990.

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

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

[Wir71] Niklaus Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

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

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

[YC79] Edward Yourdan and Larry L. Constantine. *Structured Design.* Prentice Hall, 1979.

[YTR+87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The Duality of Memory and Communication in the implementation of a Multiprocessor Operating System. In *The Proceedings of the 11th Symposium on Operating System Principles*, November 1987.

[Zit89] Martina Zitterbart. High-Speed Protocol Implementations Based on a Multiprocessor-Architecture. In *Proceedings of the 1st International Workshop on High-Speed Networks*, pages 151–163, May 1989.

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

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

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

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

[Zwe91] Jonathan M. Zweig. An Object-Oriented Framework for Implementing Network Protocols. Master's thesis, University of Illinois at Urbana-Champaign, 1991.