ADAPTIVE

A Dynamically Assembled Protocol
Transformation, Integration, and eValuation
Environment

Douglas C. Schmidt, Donald F. Box, and Tatsuya Suda
schmidt@ics.uci.edu, dbox@ics.uci.edu, and suda@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine, California 92717[1]

An earlier version of this paper appeared in the Journal of Concurrency: Practice and Experience, June 1993

Abstract

Computer communication systems must undergo significant changes to keep pace with the increasingly demanding and diverse multimedia applications that will run on the next generation of high-performance networks. To facilitate these changes, we are developing "A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment" (ADAPTIVE). ADAPTIVE provides an integrated environment for developing and experimenting with flexible transport system architectures that support lightweight and adaptive communication protocols for diverse multimedia applications running on high-performance networks. Our approach employs a collection of reusable "building block" protocol mechanisms that may be composed together automatically based upon functional specifications. The resulting protocols execute in parallel on several target platforms including shared memory and message-passing multiprocessors. ADAPTIVE provides a framework for (1) determining the functionality of customized lightweight protocol configurations that efficiently support multimedia applications and (2) mapping this functionality onto efficient parallel process architectures.

1 Introduction

This paper describes a flexible environment called the ADAPTIVE system, *"A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment"* [1, 2]. ADAPTIVE provides an integrated set of tools for developing and experimenting with flexible transport system[2] architectures that support lightweight and adaptive communication protocols. Developing efficient transport systems and protocols is becoming increasingly important to support the diverse quality-of-service (QoS) requirements of multimedia applications running over high-performance networks. For example, the throughput, latency, and reliability requirements of multimedia applications such as interactive voice, video conferencing, supercomputer visualization, collaborative work, and remote process control are more stringent and varied than those found in traditional applications such as remote login or bulk file transfer. Likewise, the channel speed, bit-error rates, diameter, and services (such as isochronous and bounded-latency delivery guarantees) of high-performance networks such as DQDB, FDDI, and ATM exceed those offered by traditional networks such as Ethernet and Token Ring.

Conventional transport systems possess several major limitations and do not adequately support multimedia applications running on high-performance networks such as DQDB, FDDI, and ATM. Among other things, these transport system limitations involve deficiencies with (1) protocol functionality, (2) protocol performance, and (3) protocol flexibility. First, existing transport systems typically provide only a small number of conventional protocols such as TCP, UDP, and RPC. These conventional protocols are limited by their extraneous and obstructing functionality and their lack of certain essential features (see Section 2 for additional details). Second, protocol performance is limited by the selection of inefficient protocol mechanisms that are not well-integrated with operating system services such as memory and process management. These performance limitations prevent multimedia applications from fully exploiting the services and channel speeds offered by the underlying high-performance networks. Third, conventional protocol are typically designed and implemented in an inflexible manner [4, 5]. This inflexibility greatly increases the effort required to customize conventional protocols to make them more suitable for particular application and high-performance network pairings.

Developing a single heavyweight protocol that efficiently supports every class of application and network appears to be infeasible [6]. One promising alternative is to devise mul-

[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. Additional support for this research was also provided by Nippon Steel Information and Communication Systems Inc. (ENICOM), Hitachi Ltd., Hitachi America, and Tokyo Electric Power Company.

[2]Transport systems combine protocol processing functionality (such as connection management, data transmission control, remote context management, and error protection) together with operating system (OS) services (such as memory management and process management) and hardware devices (such as high-speed network controllers) to support applications running on local, metropolitan, and wide area networks [3].

tiple lightweight protocols that are customized for particular pairings of application QoS requirements and network characteristics. However, the effort required to develop, deploy, and maintain each customized protocol "by hand" makes this approach impractical. Therefore, ADAPTIVE supports a more flexible approach that provides automated support for composing lightweight and adaptive protocols. ADAPTIVE is designed to improve protocol functionality and performance via the flexible transport system development and experimentation environment described below.

1.1 Overview of the ADAPTIVE System

We are designing and implementing the ADAPTIVE system to address the limitations with conventional transport systems and protocols described above. ADAPTIVE provides an integrated set of tools that enhance protocol functionality, performance, and flexibility. These tools support the synthesis of customized *lightweight protocol machines* that may be configured to execute concurrently on both shared memory and message-passing multi-processor platforms. Protocol Machines are executable "instances" of communication protocols that combine context information (such as round-trip timers, local and remote addresses, sequence numbers, and flow control window advertisements) with peer-to-peer protocol processing tasks. To enhance functionality and performance, ADAPTIVE's lightweight protocol machines are specially-tailored to contain only the context information and tasks necessary to fulfill the QoS requirements of particular applications (or classes of applications) that run in a particular network environment. This parsimonious approach reduces the time and space overhead associated with protocol processing.

ADAPTIVE also provides a controlled environment for experimenting with alternative *process architectures*. A process architecture binds operating system processes with various communication protocol entities (such as protocol layers, messages, tasks, and connections [3]). Process architectures significantly influence application performance [4]. Therefore, the lightweight protocol machines synthesized by ADAPTIVE tools may be configured so that certain tasks (such as checksum computation, segmentation, (re)transmission, and end-to-end flow control) execute in parallel on several different types of process architectures.

In addition, ADAPTIVE supports run-time reconfiguration of protocol machine functionality. This enables protocol machines to adapt dynamically to changes in application requirements (*e.g.,* switching from unreliable to reliable data delivery), transport system resources (*e.g.,* buffer space and CPU load), and network characteristics (*e.g.,* network congestion and routing). Protocol Machine adaptivity is important since applications and networks are dynamic entities that are not necessarily served most effectively by statically configured mechanisms.

To reduce the inflexibility associated with conventional protocol implementations, ADAPTIVE employs a transformational methodology that automatically generates protocol machines based upon higher-level specifications of application QoS requirements and network characteristics. These generated protocol machines are composed of fine-grain "building block" protocol processing mechanisms that reside in an object repository. This repository contains reusable implementations of various mechanisms for protocol processing such as connection establishment, retransmission, data transmission control, remote context management, demultiplexing, event timing, and message management. ADAPTIVE also provides an integrated suite of performance measurement tools. These tools monitor and analyze the run-time behavior of protocol machines in an unobtrusive and controlled manner to precisely pinpoint performance bottlenecks.

1.2 Related Work

The ADAPTIVE system is primarily influenced by the Programmable Network Prototyping System (PNPS) [7], the *x*-kernel/Avoca projects [4], the Function-based Communication SubSystem (F-CSS) [6], the Multi-Stream Protocol (MSP) [8] and the Synthesis Kernel [9].

PNPS is an environment for prototyping and experimenting with hardware implementations of MAC-layer protocols. ADAPTIVE, on the other hand, focuses on prototyping and experimenting with software architectures for middle-layer (OSI reference model layers 3 and 4) and higher-layer (layers 5-7) protocols.

The *x*-kernel is a network protocol development and experimentation environment that is hosted in various operating systems such as UNIX and Mach. It serves as a "protocol backplane" that provides a uniform interface to several reusable "layer-to-layer" protocol support tasks such as message buffering, demultiplexing, and event management. Whereas the *x*-kernel focuses primarily on operating system support for layer-to-layer protocol tasks, ADAPTIVE supports end-to-end protocol functionality such as connection handling, error protection, end-to-end transmission control, and remote context management.

Avoca uses the *x*-kernel to experiment with alternative middle-layer protocol development techniques. It emphasizes flexible "hand-crafted" implementations of protocols like RPC, UDP, and TCP that support traditional data applications running in traditional network environments. ADAPTIVE, on the other hand, focuses on automatic generation of flexible and adaptive protocol machines that support multimedia applications in high-performance network environments.

The Function-based Communication SubSystem (F-CSS) is a transport system architecture that dynamically configures executing protocol stacks based on user-specified classes of application requirements. F-CSS runs in a network of transputers that communicate via message-passing [10]. ADAPTIVE, on the other hand, also runs on general-purpose shared memory symmetric multi-processors.

The Multi-Stream Protocol (MSP) is a transport protocol that executes certain mechanisms in parallel. MSP also permits several mechanisms to change dynamically without loss of data (*e.g.,* switching the retransmission mechanism from *go-back-n* to *selective repeat* within an active protocol machine). Like the *x*-kernel, MSP focuses on mechanisms (*e.g., how* to implement the changes on-the-fly) rather than on policies that orchestrate the mechanisms (*e.g., when* to make the changes and *what* changes should occur). ADAPTIVE, on the other hand, focuses on both policies and mechanisms and also addresses OS-related issues such as process architectures.

The Synthesis Kernel is an operating system kernel that dynamically generates customized, parsimonious mechanisms for certain operating system services. For example, it generates specially-tailored, self-tuning code at run-time that minimizes scheduling and context switching overhead. Whereas the Synthesis Kernel provides adaptive mechanisms for non-distributed operating system services, ADAPTIVE provides adaptive mechanisms for distributed transport system protocol services.

The paper is organized as follows: Section 2 describes the research background, Section 3 describes several alternative process architectures supported by ADAPTIVE, Section 4 outlines ADAPTIVE's architectural design and implementation, and Section 5 summarizes the paper and outlines our future research.

2 Research Background

Transport systems must change significantly to keep pace with the increasingly diverse quality-of-service (QoS) requirements of multimedia applications that run in diverse high-performance network environments. In particular, conventional protocols (such as RPC, TCP, and TP4) supported by existing transport systems may be inadequate for multimedia application QoS requirements due to certain performance and functionality limitations. Moreover, these limitations are exacerbated by inflexible designs and implementations of conventional protocols that are difficult to modify and extend [5].

2.1 Limitations with Protocol Performance and Functionality

Conventional protocols incur unnecessary processing overhead due to (1) extraneous and obstructing functionality and (2) inefficient mechanisms. An extraneous protocol function is one that is not required to fulfill the QoS requirements of a particular application class. For example, protocols may omit the "sequence control" task for applications that do not require in-order delivery of transmitted Protocol Data Units (PDU)s (such as a distributed logging facility that collects and timestamps log records sent from multiple clients and stores them in a centralized server database). Likewise, the overhead of explicit connection management mechanisms may not be required for request-response applications (such as distributed file servers in a local area network) [11]. Moreover, strict error control may not be necessary for applications (such as full-motion video) that tolerate some degree of information loss. Extraneous functionality becomes an "obstruction" when it prevents the satisfaction of application QoS requirements. For example, the timer and buffer management overhead associated with certain retransmission mechanisms impedes real-time delivery of data for loss-tolerant, constrained latency applications such as interactive voice and video.

Conventional protocols are also limited by inefficient mechanisms that hinder application efficiency in high-performance network environments. For example, using *stop-and-wait* flow control (*e.g.,* RPC) and *go-back-n* retransmission schemes (*e.g.,* TCP and TP4) for bandwidth-sensitive applications under-utilizes the channel capacity available on high-speed and congestion-prone links, respectively. Protocol error detection and correction mechanisms for high-speed, low-error fiber optic networks may also be simplified by optimizing them for the typical case of "error-free" transmission [12].

In addition, conventional protocols often lack functionality necessary to support certain QoS requirements found in multimedia applications. For example, interactive teleconferencing applications require reliable multicast services that are not available in general-purpose protocols such as TCP or TP4. Another example is the lack of transport system support for isochronous delivery required by jitter-sensitive applications.

2.2 Limitations with Protocol Flexibility

Conventional protocols supported by existing transport systems are typically characterized by inflexible designs and implementations. This inflexibility perpetuates protocol performance and functionality limitations by making it difficult to replace inefficient mechanisms with more suitable alternatives (such as improved implementation techniques [13], connection management schemes [14], flow control algorithms [15], and error detection and recovery mechanisms [11, 16]). Therefore, many applications remain in the procrustean framework imposed by conventional protocols due to the effort and expertise required to modify these protocols without introducing subtle errors or inefficiencies [17].

The ADAPTIVE system provides tools that address the limitations of conventional protocols via an experimentation-based methodology. This approach supports (1) flexible and adaptive substitution of alternative protocol mechanisms (such as *implicit* vs. *explicit* connection establishment, *selective repeat* vs. *go-back-n* retransmission, and/or *periodic* vs. *transmission-based* remote context management) and (2) controlled measurement of the performance impact resulting these mechanism substitutions. To evaluate and experiment with alternative protocol mechanism configurations, ADAPTIVE provides a modular development environment that (1) systematically integrates and composes various protocol mechanisms that execute in several parallel process architectures and (2) unobtrusively monitors and measures protocol performance in a controlled manner.

3 Process Architectures

To address protocol performance limitations, ADAPTIVE provides an environment for experimenting with alternative *process architectures*. A process architecture binds logical and/or physical operating system processes to various communication protocol entities (such as messages, layers, tasks, and connections). The choice of a particular process architecture significantly affects application and transport system performance [4]. This section compares and contrasts the advantages and disadvantages of several process architectures supported by ADAPTIVE.

3.1 Process Architecture Models

There are two basic process architecture models: *task-based* and *Message-based*. These two models differ in terms of their structure and their performance. It is possible, however, to implement the same protocol families (such as the ISO OSI and Internet reference models) with either model.

A process architecture is a "logical" model that may or may not be implemented using multiple *processing elements* (PEs). Several different approaches have been proposed to map Task-based and Message-based process architectures onto multiple PEs [18, 19, 10, 20, 21]. Figures 1 and 2 illustrate four models of process architecture parallelism: *Layer Parallelism* and *Task Parallelism* are Task-based process architectures; *Connectional Parallelism* and *Message Parallelism* are Message-based process architectures. As described in Section 4.2, the ADAPTIVE system supports all these models. Several criteria used to evaluate the different parallelism models include (1) the level of parallelization supported, (2) the overhead of synchronizing the PEs, (3) the interprocess communication overhead associated with passing messages between the PEs, and (4) whether load balancing is supported effectively.

Task-based Process Architectures: Task-based process architectures associate OS processes with layers or protocol tasks, rather than with messages or connections.

- **Layer Parallelism:** Layer Parallelism is a coarse-grained Task-based process architecture. As shown in Figure 1 (1), a PE is associated with each protocol layer in the protocol stack. Messages flow through the layers in a coarse-grain "pipelined" manner. Inter-layer buffering and flow control is typically necessary since processing at each layer may not execute at the same rate. The primary advantage of layer parallelism is the simplicity of its design. The primary disadvantages are (1) its fixed amount of parallelism (limited by the number of protocol layers), (2) potentially high synchronization and communication overhead (*e.g.,* the cost of synchronization and moving messages between layers), and (3) poor support for PE load balancing (PEs are dedicated to specific protocol layers).

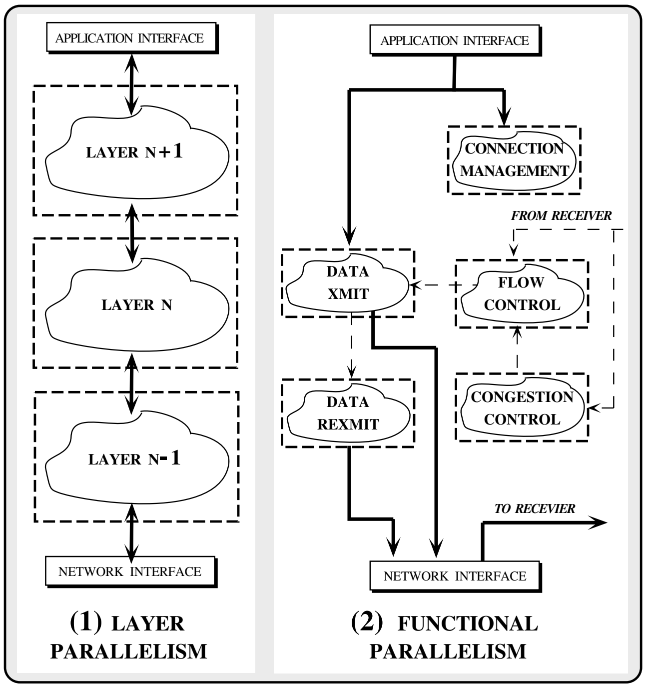

Figure 1: Task-based Process Architectures

- **Task Parallelism:** Task Parallelism is a fine-grained Task-based process architecture. This approach utilizes multiple PEs that perform protocol processing tasks in parallel. These tasks include (1) connection management, (2) header composition and decomposition (*e.g.,* address resolution and demultiplexing), (3) PDU-level and bit-level error protection (*e.g.,* detecting and reporting out-of-sequence PDUs and performing checksum computation), (4) segmentation and reassembly, and (5) flow control. Figure 1 (2) illustrates a Task Parallelism configuration where multiple PEs operate as a fine-grain pipeline on messages flowing through the sender-side of a protocol machine. Designs based on Task Parallelism often assume a "delayered" protocol stack that relaxes the protocol layer boundaries associated with the Internet and ISO OSI reference models [18]. The primary advantage of this approach is the performance gain from using multiple PEs. However, the disadvantages are that it is difficult to eliminate the memory contention, synchronization, and interprocess communication overhead. Moreover, as with Layer Parallelism, Task Parallelism does not facilitate load balancing.

Task-based process architectures have several advantages. For example, this approach often corresponds closely to standard layered communication architecture specifications, which helps to simplify protocol family designs and implementations [22]. Moreover, each protocol component performs its processing within a single address space. This provides an implicit serialization point for messages destined for the same protocol component, thereby reducing the effort required to implement synchronization and mutual exclusion logic.

However, Task-based process architectures have several disadvantages. For example, depending on the operating system structure and underlying hardware, they may incur a high amount of context switching, scheduling, synchronization and data copying as messages flow through the layered protocol components [3]. In addition, the parallelism provided by a Task-based process architecture may be limited if only a one-to-one correspondence exists between processes and protocol layers. In particular, most existing protocol families (such as OSI and TCP/IP) possess relatively few protocol layers.

Message-based Process Architectures: Message-based process architectures associate OS processes with messages and connections rather than with protocol layers or tasks [4, 20].

- **Connectional Parallelism:** Connectional Parallelism is a Message-based process architecture that dedicates a separate PE for each active connection. Figure 2 (1) illustrates this approach, where connections C_1, C_2, C_3, and C_4 are bound to separate PEs that process all messages associated with their connection. This approach is useful for servers that maintain many open connections simultaneously. The advantages of Connectional Parallelism are (1) the inter-layer communication overhead is reduced (since moving between protocol layers may not require switching the process context), (2) the synchronization and communication overhead is relatively low within a given connection, and (3) the degree of parallelism is a function of the number of active connections rather than the number of layers. One disadvantage with Connectional Parallelism is the difficulty of PE load balancing. For example, a highly active connection may swamp its PE with messages, leaving other PEs tied up at less active or idle connections. In addition, it is typically necessary to use *packet filters* [23] at the network interface. Packet filters allow higher-level protocols to instruct the network interface to demultiplex particular types of PDUs to them. Packet filters are necessary for Connectional Parallelism since the network interface must demultiplex using PDU address information (such as connection identifiers or port numbers) that is actually associated with protocols several layers above in the protocol stack.

- **Message Parallelism:** Figure 2 (2) depicts Message Parallelism, where a separate PE is associated with each incoming or outgoing message. These messages are typically stored in shared memory buffers. A pointer to the message is passed to the next available PE, which performs all the protocol processing tasks on that message. The advantages of Message Parallelism are similar to Connectional Parallelism. Moreover, the level of parallelism may be very high since it is a function of the number of messages exchanged, rather than the number of layers, tasks, or connections. In addition, processing loads may be balanced more evenly between PEs. The primary disadvantage of Message Parallelism is the overhead of synchronizing messages bound for the same higher-layer protocol machine. This overhead results from synchronization and mutual exclusion primitives that serialize access to shared resources (such as memory buffers and protocol machine control blocks used to reassemble protocol segments bound for the same higher-layer protocol machine).

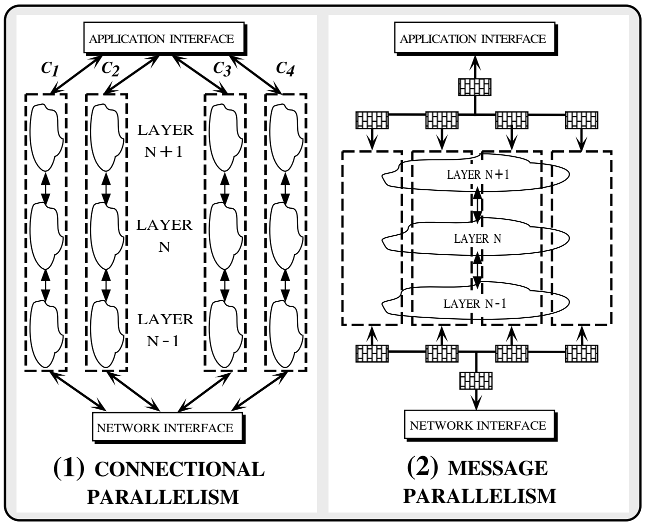

Figure 2: Message-based Process Architectures

Message-based process architectures have several advantages. For example, they may effectively use a large number of available processing elements if processes are associated with messages [20]. This increased parallelism may improve load balancing, leading to higher overall transport system throughput. For example, each incoming message may be dispatched to an available processing element on a massively parallel multi-processor. Moreover, since all the protocol tasks applied to a message reside in the same process address space, synchronous *intra-process* subroutine calls and upcalls [24] may be used to communicate between the protocol layers. This may be more efficient than using asynchronous *inter-process* message queues, which incur additional context switch overhead [4] when exchanging messages between layers in different processes. In addition, Message-based process architectures do not impose a total ordering on messages destined for the same protocol machine. This is advantageous for protocols that utilize *application level framing* [25], where application data unit boundaries are maintained throughout the layered protocol processing stages.

Message-based process architectures also have several disadvantages. For instance, performance may degrade significantly if the OS is incapable of associating a process with each message efficiently. This problem is exacerbated when communication loads are high and message inter-arrival and departure times are close together. In addition, complex interactions between messages and protocol machines may increase synchronization complexity, mutual exclusion overhead, and shared memory contention at the receiver. For example, messages bound for the same higher-layer protocol machines must coordinate to share protocol machine state information correctly between multiple cooperating processes.

4 Design of the ADAPTIVE System

The research objectives of the ADAPTIVE project are (1) to precisely specify and classify application QoS requirements, (2) to determine appropriate process architecture and protocol mechanisms that support these QoS requirements in various high-performance network environments, (3) to precisely characterize the interfaces and interrelationships between the mechanisms, (4) to develop object-oriented techniques for designing, implementing, and composing these mechanisms efficiently, correctly, and modularly, (5) to integrate the mechanisms into suites of customized lightweight protocols that support application and network diversity, and (6) unobtrusively collect metrics that measure protocol machine behavior and precisely pinpoint performance bottlenecks. The following section summarizes the major components of the ADAPTIVE system developed to meet these objectives.

4.1 ADAPTIVE Components and Transformational Phases

The two primary stages of ADAPTIVE are *protocol machine generation* and *protocol machine execution*. In the protocol machine generation stage, ADAPTIVE creates executable lightweight protocol machines. These protocol machines are customized for the QOS requirements of particular applications (or classes of applications) that run in a particular network environment. In the protocol machine execution stage, applications execute generated protocol machines to perform their data transport activities efficiently. If the preconfigured protocol machines are inadequate, applications may also customize protocol machine functionality at runtime using additional ADAPTIVE reconfiguration services.

The architecture of the ADAPTIVE system contains several components used during these stages. These components are represented via a collection of formalisms, tools, and resources that perform various activities related to protocol machine generation, execution, and measurement. The remainder of this subsection discusses these components, formalisms, tools, and activities.

4.1.1 Protocol Machine Generation Stage

To facilitate automation and reuse of various ADAPTIVE system services and tools, the protocol machine generation stage is organized into several distinct phases. Figure 3 illustrates the transformations between the primary system components in each phase. The first transformation turns specifications of application QoS requirements into platform-independent Protocol Machine configurations. These configurations describe the processing tasks that will be executed in a particular order on incoming and outgoing PDUs. The second transformation turns the resulting configurations into Protocol Machine instantiations that are suitable for the target execution platform. These instantiations are derived from a collection of reusable protocol mechanisms stored in an object repository.

(A) Protocol Machine Specification Phase: In the specification phase, ADAPTIVE's transformation components receive descriptions of application QoS requirements. These components then attempt to produce executable instantiations of protocols that correspond to the specified requirements. Specifications of QoS requirements are passed to the ADAPTIVE system either *statically* (*e.g.,* by system configuration utilities during transport system boot-time) or *dynamically* (*e.g.,* by applications during their connection initiation phase). QoS requirements may be specified via several interfaces:

- **Protocol Machine Specification Language:** This is a high-level, non-procedural notation that specifies *quantitative* and *qualitative* application QoS requirements [6]. Quantitative criteria represent "measurable" characteristics of QoS such as "bit per-second," "milli-seconds," or "errors per PDU." Qualitative criteria represent attributes such as "reliable," "in-order delivery," and "record-oriented." As described below in the protocol machine configuration phase, application QoS requirements written in the Protocol Machine Specification Language are submitted to the Protocol Machine Configurator, which converts these requests into a lower-level platform-independent notation.

- **Named Protocol Machine Selection:** Applications may bypass the bulk of the configuration and instantiation process via a Named Protocol Machine Selection facility that directly invokes preconfigured Protocol Machine Instantiations stored in a system library. This facility minimizes the run-time performance overhead associated with the protocol machine generation stage. It is used by applications whose QoS requirements are satisfied by existing preconfigured protocol machines for standard network services (such as file transfer, remote login, or interactive voice). The choice of using the Protocol Machine Specification Language vs. the Named Protocol Machine Selection involves a tradeoff between customized functionality and reduced protocol machine generation overhead.

(B) Protocol Machine Configuration Phase: The configuration phase transforms Protocol Machine Specification Language descriptions into platform-independent Protocol Machine Task Graph Configuration Language descriptions. A Protocol Machine Task Graph is an abstraction that represents a "blueprint" of protocol machine functionality. It describes the peer-to-peer tasks (such as connection management, segmentation, or duplicate control) and ordered interrelations (such as "perform resequencing before reassembly" or "compute checksum before flow control") of various tasks that process the PDUs. As shown in Figure 4, each node in

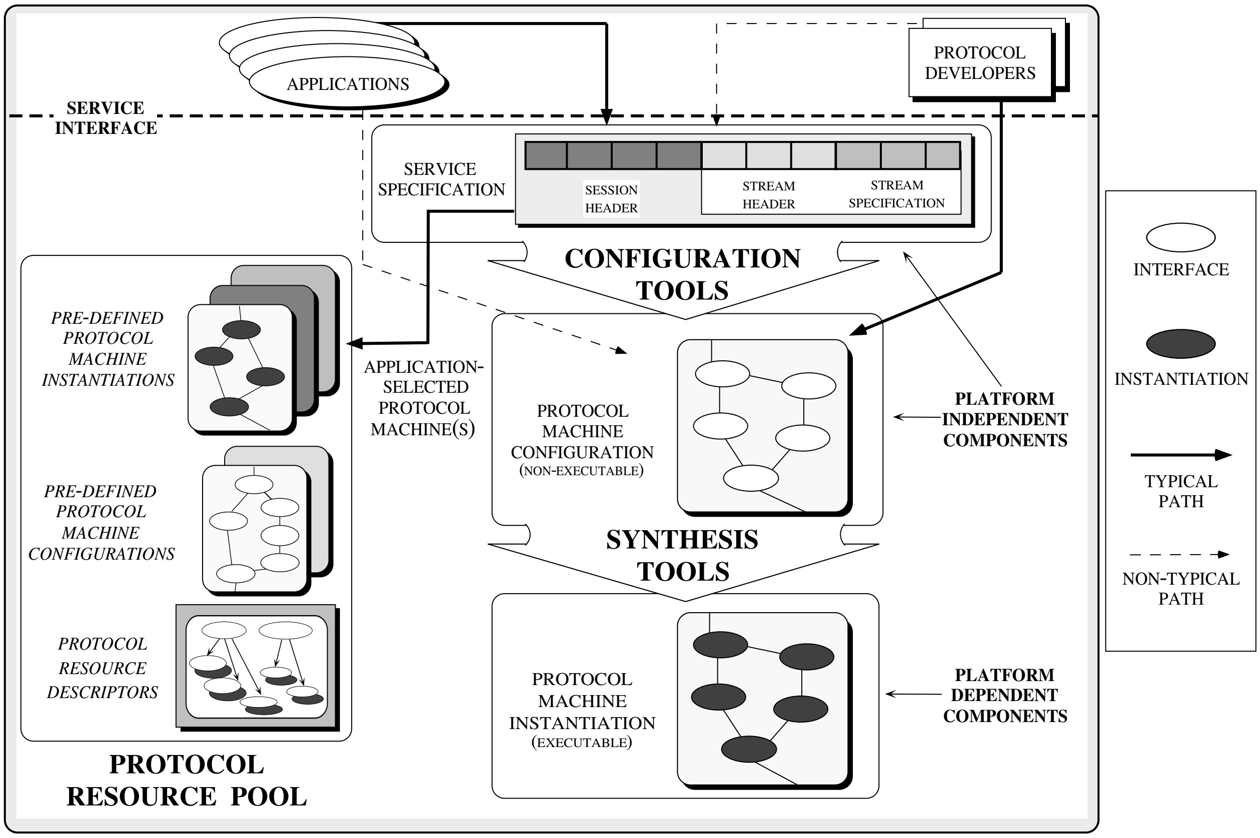

Figure 3: ADAPTIVE System Components and Transformations

the graph constitutes a well-defined task such as connection establishment and termination, retransmission, acknowledgment, flow control, checksum calculation, multicast, or routing. The edges of the graph express interrelations between the nodes and characterize (1) the flow of PDU control information and (2) the relationships between the ordered tasks. The configuration phase generates Protocol Machine Task Graph Configurations via the following tools and formalisms:

- **Protocol Machine Configurator:** This tool transforms specifications of application QoS requirements written in the Protocol Machine Specification Language into a Protocol Machine Task Graph Configuration. As with the protocol machine specification phase, input to the Protocol Machine Configurator may be submitted either *statically* (during transport system boot-time) or *dynamically* (at run-time during connection initiation). The Protocol Machine Configuration generates a "program" written in the Protocol Machine Task Graph Configuration Language described in the following bullet. As shown in Figure 3, applications and system configurations utilities may bypass the Protocol Machine Configurator and specify the Protocol Machine Task Graph Configuration directly. This "short-circuit" interface offers a tradeoff between the convenience of programming at a higher-level of abstraction and increased control over protocol functionality.

- **Protocol Machine Task Graph Configuration Language:** This language is a platform-independent notation used to configure a particular set of mechanisms that will be executed to process incoming and outgoing PDUs. As described in the protocol machine instantiation phase below, programs written in this language are submitted to the Protocol Machine Synthesizer, which produces Protocol Machine Instantiations created from mechanisms stored in a Protocol Machine and Protocol Mechanism Repository. The Protocol Machine Task Graph Configuration Language represents a lower-level of abstraction than the Protocol Machine Specification Language (similar to the difference between programming in an assembly language vs. a fourth-generation language).

(C) Protocol Machine Instantiation Phase: The instantiation phase transforms platform-independent configurations written in the Protocol Machine Task Graph Configuration Language into Protocol Machine Instantiations that are executable on a particular target platform such as a shared memory multi-processor or a network of message-passing transputers. The instantiation phase utilizes the following tools and resources:

- **Protocol Mechanism Repository:** This repository contains reusable implementations of various schemes for connection establishment, retransmission, data transmission control, remote context management, demultiplexing, event

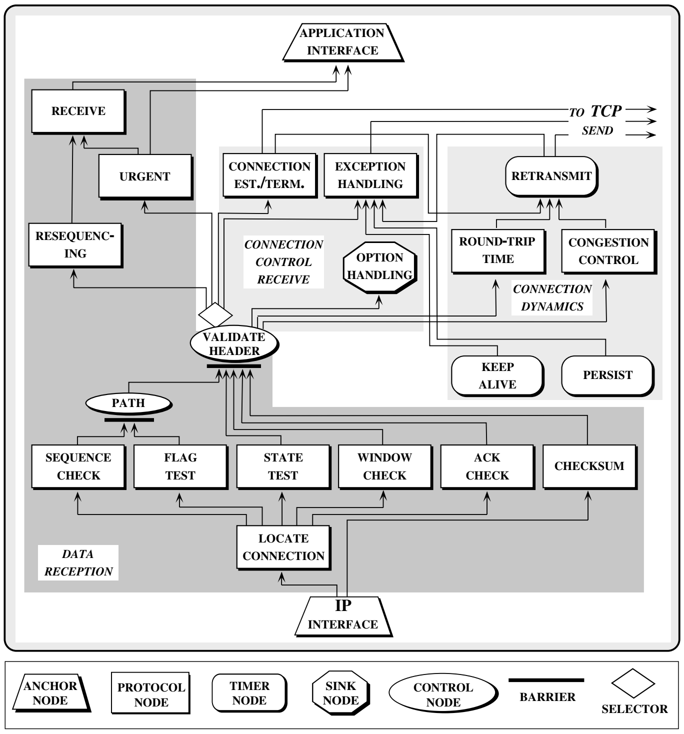

Figure 4: A Sample Protocol Machine Task Graph

timing, and message management. Repository mechanisms are written in C++ using object-oriented design and implementation techniques such as data abstraction, inheritance, and dynamic binding. These techniques reduce the effort required to develop modular, flexible, extensible, and efficient transport system software. For example, to enhance the modularity of protocol mechanisms, C++'s data abstraction features are used to integrate protocol machine context information together with the associated protocol mechanisms. Moreover, C++ inheritance feature supports flexible protocol machine composition by sharing and reusing existing protocol mechanisms. In addition, dynamic binding enhances extensibility and efficiency by deferring certain representation decisions until run-time, when addition information is available to guide the selection of more efficient protocol mechanisms.

• **Protocol Machine Synthesizer:** This tool transforms the Protocol Machine Task Graph Configuration into a Protocol Machine Instantiation. This instantiation is produced by linking together certain C++ objects in the Protocol Mechanism Repository to produce an executable representation that optimized for a particular target platform. The activities performed by the Protocol Machine Synthesizer (such as syntactic and semantic analysis, optimization, and code generation) are similar to those used for compiling high-level programming languages into object code. The Protocol Machine Synthesizer is designed to work either in conjunction with the Protocol Machine Configurator or as a stand-alone tool.

• **Protocol Machine Instantiations and Data Streams:** Protocol Machine Instantiations orchestrate the interaction of one or more Data Streams. Data Streams are executable representations possessing protocol mechanisms that support a particular set of QoS requirements during a specified time period. As illustrated in Figure 5, Data Streams are composed of protocol mechanisms that are customized for particular application QoS requirements and network capabilities. Moreover, since applications may have different QoS requirements for their sender and receiver sides, each Data Stream is uni-directional. For example, a file transfer application may be implemented via a Protocol Machine Instantiation possessing two uni-directional Data Streams with different QoS characteristics for sending and receiving control and data PDUs. Protocol Machine Instantiations also provide a synchronization point for multimedia applications that exchange multiple types of related Data Streams (such as separate voice, video, and text channels in a multimedia tele-conference). In ADAPTIVE, Data Stream functionality may be specified via the Protocol Machine Task Graph Configuration Language and automatically synthesized from C++ objects residing in the Protocol Mechanism Repository described above.

4.1.2 Protocol Machine Execution Stage

During the protocol machine execution stage, applications transfer data using Protocol Machine Instantiations created in the protocol machine generation stage. The Protocol Machine Task Engine is the primary tool used during the protocol machine execution stage:

• **Protocol Machine Task Engine:** When an application "opens" a Protocol Machine Instantiation, the Protocol Machine Task Engine dynamically loads and invokes the appropriate Protocol Machine Instantiation(s) in the ADAPTIVE run-time environment. This invocation sequence allocates and initializes the necessary protocol machine control blocks and system data structures, links protocol mechanisms together, and associates these mechanisms with one or more operating system processes (note that these processes may be mapped onto logical and/or physical hardware processing elements).

ADAPTIVE optionally collects application and protocol machine performance metrics during the execution stage. These metrics quantify the performance tradeoffs that result from selecting different Protocol Machine Instantiations to support application QoS requirements. This, in turn, enables meaningful analysis and evaluation of alternative design and implementation strategies and helps tune transport system components and parameters to improve performance.

A variety of metrics are required to characterize the performance of protocols that support multimedia applications. ADAPTIVE metrics are divided into two classes: *blackbox* and *whitebox*. These classes differ depending on whether the metric collection mechanisms instrument the internals of Protocol Machine Instantiations. Blackbox metrics (such

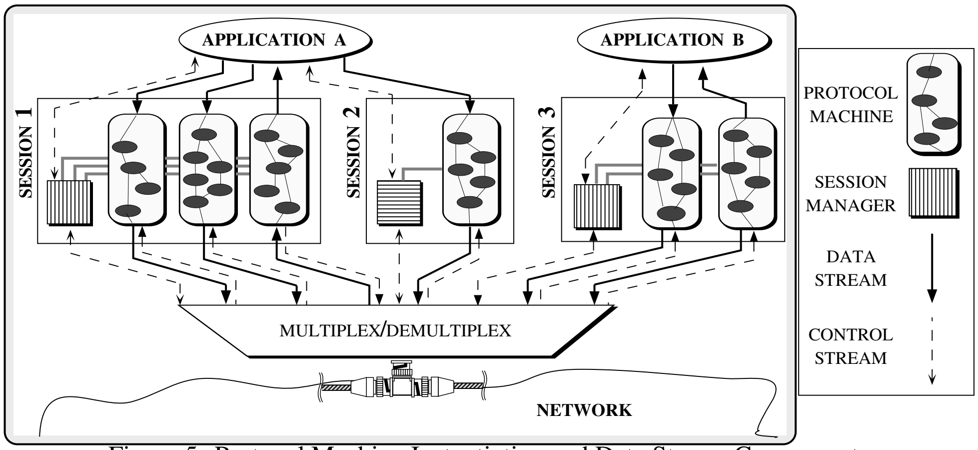

Figure 5: Protocol Machine Instantiation and Data Stream Components

as throughput and average end-to-end delay) may be collected *without* instrumenting the Protocol Machine Instantiations, whereas Whitebox metrics typically require some form of instrumentation. Whitebox metrics include connection establishment and termination latency, number of PDU (re)transmissions, the number of instructions required to execute a particular protocol task, interrupt and scheduling overhead, the degree of jitter, and PDU loss. Note that without a development and testing environment such as ADAPTIVE, it is difficult to collect white-box metrics and perform fine-grained experimentation in an unobtrusive and controlled manner.

ADAPTIVE provides flexible and extensible tools that calculate standard metrics and incorporate new metrics. Existing research has explored various data collection techniques such as software vs. hardware monitoring, time-driven vs. event-driven monitoring, and code profiling. ADAPTIVE integrates these approaches to extract their benefits and minimize their run-time overhead. The collected metric data is analyzed using techniques such as critical path analysis and may be displayed on a per-protocol machine, per-host, per-subnetwork basis. The following tools and interfaces are used to measure and analyze protocol machine performance:

- **Metric Transceiver Daemon:** The Metric Transceiver Daemon receives metric samples generated by instrumented Protocol Machine Instantiations executing in the ADAPTIVE system. The Transceiver optionally performs certain preprocessing operations (such as generating new samples with information aggregated from smaller samples, filtering out unneeded samples, etc.) and forwards these samples to the Metric Collector Daemon.

- **Metric Collector Daemon:** The Metric Collector Daemon receives metric samples from the Metric Transceiver Daemon. It computes metric values in a periodic or event-driven manner and displays them via a textual or graphical interface. In addition, the Metric Collector Daemon employs a promiscuous network packet filter [23] to examine all packets in the network and unobtrusively collect the appropriate metric samples.

- **Metric Collection Interface:** This interface enables protocol developers to instrument their code with "software markers." These markers generate time-stamped metric samples that report information such as the size and sequence number of transmitted and received PDUs. These samples are forwarded automatically to the Metric Transceiver Daemon.

4.2 Process Architecture Support

The existing ADAPTIVE prototype is written in user-space. It provides Connectional Parallelism via the multi-processing capabilities of a 4-CPU SPARC Server 690MP multi-processor running SunOS 4.1.2. Each connection runs on its own processor as long as the number of connections is less than the number of CPUs. To experiment with several other process architecture environments, we are porting the prototype to the *x*-kernel [4] and System V STREAMS [26]. This section briefly outlines these systems and describes how ADAPTIVE is integrated into the various process architectures.

The *x*-kernel employs a "process-per-message" Message-based process architecture that may be configured to run inside the host OS kernel and/or in user-space. When a message arrives at a network interface, a separate process is dispatched from a pool of lightweight threads to escort the message upwards through the appropriate chain of protocol and protocol machine objects. In general, only one context switch is required to shepard a message through the protocol stack, regardless of the number of intervening protocol layers.

System V STREAMS provides services for supporting several process architectures including Layer Parallelism, Task Parallelism, and Connectional Parallelism. In the STREAMS approach, PDUs flows through a bi-directional stack of STREAM modules that are located between an application process and a network interface. STREAM modules perform protocol processing operations on the data they receive and then forward the data to an adjacent module. Each module contains a read queue and a write queue that implement the module's processing operations and regulate layer-to-layer message flow between adjacent queues. In many STREAMS implementations (such as OSF and Solaris), separate lightweight processes may be associated with the STREAM module's read and write queues. These processes may be configured to work in a "pipeline fashion," performing various protocol tasks on incoming and outgoing PDUs in parallel.

The Protocol Machine Instantiation and Data Stream components of ADAPTIVE are implemented to be relatively independent of the process architecture provided by the host operating system. By leveraging off several C++ mechanisms (such as inheritance and dynamic binding, transparent free store management, member function inlining, and conditional compilation), the ADAPTIVE subsystems possess a small set of well-defined dependencies on the underlying process architecture. This facilitates controlled experimentation with different process/protocol decomposition schemes and provides additional transparency to protocol developers. For example, objects that are accessed via several processes are allocated in shared memory segments. This allows multiple threads of control (executing in distinct address spaces) to inspect and/or modify the shared data structures. Objects that are shared between multiple processes are conditionally compiled to include the necessary mutual exclusion code to synchronize the multiple threads of control.

5 Summary

ADAPTIVE is a flexible transport system development and experimentation environment designed to address the increasingly diverse quality-of-service requirements of multimedia applications running on high-performance networks. ADAPTIVE supports diverse applications and networks via (1) its customized lightweight and adaptive protocol machines and (2) its alternative process architectures that help improve protocol performance and reduce transport system overhead. In addition, ADAPTIVE's architecture facilitates an "experimentation-based" protocol development methodology based on feedback-guided monitoring and measurement. This enables ADAPTIVE to precisely measure the performance effects resulting from selective modification of certain process architecture and protocol mechanisms.

We are currently designing and implementing a prototype implementation of ADAPTIVE written in C++. To experiment with alternative process architectures, this prototype is being ported to the *x*-kernel's Message-based process architecture, as well as to System V STREAMS, which supports both Message-based and Task-based process architectures. We are using the prototype to experiment with alternative process architectures and communication protocols that support multimedia applications (such as network voice and video) running on several different networks (such as Ethernet, DQDB, and FDDI).

References

[1] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A Flexible and Adaptive Transport System Architecture to Support Lightweight Protocols for Multimedia Applications on High-Speed Networks," in *Proceedings of the* 1^{st} *Symposium on High-Performance Distributed Computing (HPDC-1)*, (Syracuse, New York), pp. 174–186, IEEE, September 1992.

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

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

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

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

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

[7] R. C. Albert Li, A. Fawaz, S. Sachs, P. Varaiya, and J. Walrand, "The Programmable Network Prototyping System," *IEEE Computer*, vol. 22, pp. 67–76, May 1989.

[8] T. F. L. Porta and M. Schwartz, "Design, Verification, and Analysis of a High Speed Protocol Parallel Implementation Architecture," in *Proceedings of the First IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Feb. 1992.

[9] C. Pu, H. Massalin, and J. Ioannidis, "The Synthesis kernel," *Computing Systems*, vol. 1, pp. 11–32, Winter 1988.

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

[11] D. R. Cheriton, "VMTP: Versatile Message Transaction Protocol Specification," *Network Information Center RFC 1045*, pp. 1–123, Feb. 1988.

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

[13] V. Jacobson, "Congestion Avoidance and Control," in *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, (Stanford, Calif.), August 1988.

[14] R. W. Watson, "The Delta-*t* Transport Protocol: Features and Experience," in *Proceedings of the 1st International Workshop on High-Speed Networks*, May 1989.

[15] D. D. Clark, M. L. Lambert, and L. Zhang, "NETBLT: A Bulk Data Transfer Protocol," *Network Information Center RFC 998*, pp. 1–21, Mar. 1987.

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

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

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

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

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

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

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

[23] J. C. Mogul, R. F. Rashid, and M. 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.

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

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

[26] SunSoft, *SunOS 5.1 STREAMS Programmer's Guide*. SunSoft, 1992.