# ADAPTIVE

## A Flexible and Adaptive Transport System Architecture to Support Lightweight Protocols for Multimedia Applications on High-Performance Networks

Douglas C. Schmidt, Donald F. Box, and Tatsuya Suda
Department of Information and Computer Science
University of California, Irvine, California 92717 [1]

## Abstract

*Transport systems integrate operating system services such as memory and process management together with communication protocols that utilize these OS services to support distributed applications running on local and wide area networks. Existing transport systems do not customize their services to meet the quality-of-service requirements of distributed applications. This often forces developers into the procrustean framework offered by transport systems and protocols that were designed before the advent of multimedia applications and high-performance networks. Flexible and adaptive transport systems, on the other hand, dynamically configure lightweight protocols that meet application requirements more precisely.*

*This paper describes a high-performance transport system architecture called ADAPTIVE, "A Dynamically Assembled Protocol Transformation, Integration, and Validation Environment." We are developing ADAPTIVE to support multimedia applications running on high-performance networks. ADAPTIVE is a transformational system that provides policies and mechanisms to automatically specify and synthesize a flexible, lightweight, and adaptive transport system configuration. In addition, it provides a controlled prototyping environment for monitoring, analyzing, and experimenting with the performance effects of alternative transport system designs and implementations.*

# 1 Introduction

The performance of distributed applications such as remote terminal access, network file servers, distributed databases, bulk data transfer, computer imaging, tele-conferencing, full-motion video, scientific computation and visualization is influenced by both network factors and transport system factors. Several key network factors include channel speed, bit-error rate, and congestion levels at the intermediate switching nodes. Transport system factors include both protocol processing activities (such as connection management, transmission control, and reliability management) and general operating system hardware and software factors (such as memory latency, CPU speed, interrupt and context switching overhead, process architecture, and message buffering). Application performance bottlenecks are shifting from the network factors to the transport system factors due to (1) advances in fiber optics and VLSI technology that have increased network channel speed by several orders of magnitude and (2) the increased diversity and dynamism in application requirements and network characteristics.

To handle these changes, transport systems must become more flexible, lightweight, and adaptive [1]. However, existing transport systems typically offer only a small number of "monolithic" protocols. Moreover, these protocols do not adequately meet the communication requirements of next-generation distributed applications [2]. We are developing the ADAPTIVE system to provide a flexible architecture for developing and experimenting with lightweight and adaptive transport system protocols.

The paper is organized as follows: Section 2 describes the research background, Section 3 introduces the ADAPTIVE system, Section 4 outlines ADAPTIVE's architectural design, and Section 5 summarizes the paper and describes our future work.

# 2 Background

This section defines three important research problems involving transport systems, describes how existing systems do not adequately solve these problems, and compares ADAPTIVE with other related work.

## 2.1 Research Problems

Our research is investigating solutions to the following three problems involving transport systems:

**(A) The Throughput Preservation Problem:** As noted by [3, 4], only a limited amount of the available bandwidth in high-performance networks is being delivered to applications. This situation results both from poorly layered architectures [5] and from transport system overhead such as memory-to-memory copying and process management operations like process/context switching and scheduling. Moreover, this overhead is not decreasing as rapidly as the network channel-speed is increasing, due in part to

[1] An earlier version of this paper appeared in the proceedings of the first symposium on High-Performance Distributed Computing in Syracuse, New York, September 1992.

| Transport Service Class | Example Applications | Average Thruput | Burst Factor | Delay Sens | Jitter Sens | Order Sens | Loss Tolerance | Priority Delivery | Multi-cast |
|---|---|---|---|---|---|---|---|---|---|
| **Interactive Isochronous** | Voice Conversation | low | low | high | high | low | high | no | no |
| | Tele-Conferencing | mod | mod | high | high | low | mod | yes | yes |
| **Distributional Isochronous** | Full-Motion Video (comp) | high | high | high | mod | low | mod | yes | yes |
| | Full-Motion Video (raw) | very-high | low | high | high | low | mod | yes | yes |
| **Real-Time Non-Isochronous** | Manufacturing Control | mod | mod | high | var | high | low | yes | yes |
| **Non-Real-Time Non-Isochronous** | File Transfer | mod | low | low | N/D | high | none | no | no |
| | TELNET | very-low | high | high | low | high | none | yes | no |
| | On-Line Transaction Processing | low | high | high | low | var | none | no | no |
| | Remote File Service | low | high | high | low | var | none | no | yes |

Table 1: Application Transport Service Classes (TSC)

the widespread use of traditional transport system infrastructures such as UNIX, which are ill-suited for asynchronous, interrupt-driven network communication. Transport system overhead also affects other network characteristics such as delay and jitter.

**(B) The Application and Network Diversity and Dynamism Problem:** Another challenge facing transport system developers and researchers is how to effectively support the diverse quality-of-service requirements of next-generation distributed multimedia applications (we will refer to these as simply "application requirements" from now on) running over diverse, high-performance networks.

**Multimedia Applications:** Multimedia application requirements are more demanding and dynamic than those found in traditional applications like remote terminal access, file transfer, and electronic mail. For example, multimedia applications involve combinations of quality-of-service requirements such as extremely high throughput (full-motion video), strict real-time delivery (manufacturing control systems), low latency (on-line transaction processing), low delay jitter (voice conversation), capabilities for multicast (collaborative work activities) and broadcast (distributed name resolution), high-reliability (transfer of medical images), temporal synchronization (tele-conferencing), and some degree of loss tolerance (hierarchically-coded video).

Multimedia applications also impose different network traffic patterns. For instance, some applications generate highly bursty traffic (variable bit-rate video), some generate continuous traffic (constant bit-rate video), and others generate short, interactive, "request-response" traffic (network file systems using remote procedure calls (RPC)). In addition, application requirements may change dynamically during a session (*e.g.,* a tele-conferencing application may switch between unicast and multicast as participants join and leave the conversation). Table 1 illustrates the diversity of transport requirements for several representative classes of distributed applications [6].

**High-Performance Networks:** Network diversity involves both the static architecture and dynamic state of the underlying network. A diverse collection of networks now exists, possessing differences in *diameter* (LANs, MANs, and WANs), *channel speeds* (4/16 Mbps Token Ring, 10 Mbps Ethernet, 100 Mbps FDDI, and 155/622 Mbps ATM), *channel bit-error rate* (approximately $10^{-4}$ for copper and $10^{-9}$ for fiber), *network type* (datagram, virtual circuit, multicast, and broadcast), *media access control scheme* (CSMA/CD, token passing, and switch based), *maximum transmission unit* (48 bytes for ATM and DQDB cells, 1,500 bytes for Ethernet packets, 4,500 bytes for an FDDI frame, and 9,188 bytes for SMDS) and *network services* (synchronous, asynchronous, and isochronous delivery and bounded-delay guarantees).

Moreover, given the large installed network base, many applications will continue to interoperate across several environments in the foreseeable future. These environments include (1) low-utilization, low-latency LANs (*e.g.,* Ethernet), (2) congestion-prone, high latency WANs (*e.g.,* the current Internet), and (3) high-bandwidth, high latency WANs (*e.g.,* ATM-based B-ISDN public access networks) [2]. Handling this diversity and dynamism requires determining appropriate end-to-end congestion and error protection schemes. These schemes must effectively utilize the high bandwidth channels and adapt quickly to dynamically changing network conditions such as congestion or routing updates.

**(C) The Communication Software Development Complexity Problem:** Communication software has traditionally been difficult to develop, due to the complexity of trying to simultaneously maximize performance, functionality, correctness, extensibility, and portability across heterogeneous operating environments. This heterogeneity is reflected in complex software architectures that must support diverse applications written in diverse languages, communicating via diverse transport systems running on diverse computer architectures, which interoperate over diverse internetworked environments. To effectively support this heterogeneity, it is increasingly important to structure communication software using modular designs that enhance maintainability, flexibility, and efficiency. In particular, complicated communication software increases the difficulty of migrating selected transport system operations into more efficient VLSI hardware implementations.

## 2.2 Inadequacy of Existing Transport System Solutions

Existing transport systems do not provide effective solutions for all three problems described above. This section describes several reasons for the inadequacy of existing systems.

**(A) High Transport System Overhead:** Effectively supporting application requirements involves suitable performance and services from both the network and the transport system. However, most general-purpose operating systems do not provide adequate support for transport system activities such as real-time process scheduling, interrupt handling and context switching, message buffer management, layer-to-layer flow control, and multiplexing and demultiplexing [7]. Moreover, many protocol suites are implemented with improperly layered process architectures that increase protocol processing overhead [8].

Due to this overhead, the bandwidth available in a high-performance network is reduced by 1 to 2 orders of magnitude by the time it is actually delivered to applications [4]. Moreover, this throughput preservation problem persists despite an increase in CPU speeds. There are several explanations for this behavior: (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) [3], (2) existing transport systems are written largely in software, (3) network host interfaces typically generate interrupts for every transmitted and received packet, leading to increased CPU context/process switching overhead [3, 9], and (4) despite increasing total MIPS, RISC architectures often penalize interrupt-driven network communication by exhibiting high context/process switching overhead. This overhead results from the cost of flushing instruction and data caches and pipelines, as well as from storing and retrieving register windows and invalidating translation-lookaside buffers.

**(B) Inflexible and Non-Adaptive Transport System Architectures:** Existing transport systems are also unable to meet the diverse and dynamic demands of applications and networks since they typically provide only a fixed, statically configured suite of protocol services [2]. For instance, most general-purpose protocols are not capable of dynamically tailoring their services to take advantage of specific application requirements or network characteristics [10]. Moreover, communication software tends to be inflexible and difficult to modify [2], and is often tightly coupled to a particular transport system environment such as TCP/IP on BSD UNIX [7, 11]. All these factors increase the difficulty of maintaining, extending, and adapting the software to support the increased diversity in networks and applications.

For example, it is difficult to select an existing transport system configuration that is neither too *overweight* nor too *underweight* to satisfy the QoS requirements. An example of an overweight configuration is one where a protocol (such as TP4) provides retransmission support for loss-tolerant, constrained latency applications such as interactive voice. In this case the extra mechanisms required to provide retransmission simply slow down the protocol processing. An example of an underweight configuration is one where a protocol (such as TCP) does not provide a service (such as reliable multicast support) for applications that require it (such as interactive teleconferencing applications).

To illustrate the range in the design space, transport systems are distinguished below by how they adapt their configurations in response to dynamically changing application requirements and network characteristics:

- **Static Transport Systems:** are completely configured at operating system boot time. In this case, the transport system often provides protocol support based only upon general usage assumptions such as *unreliable datagram service* (*e.g.,* UDP) versus *reliable byte stream service* (*e.g.,* TCP). Many well-known transport systems belong to this category [12, 13, 14, 15, 16].

- **Dynamic Transport Systems:** postpone complete configuration until either connection establishment time or during data transfer. In both cases, these transport systems may be configured by negotiating with (1) local and remote hosts and (2) intermediate switching nodes in the network. This negotiation process customizes transport system sessions to account for the application requirements and the available host and network resources. These resources include available buffer space, CPU load, available communication ports and virtual circuits, and network congestion level.

Only a handful of transport systems provide any support for configuration of their protocol suites at connection establishment time [2, 17, 18]. Furthermore, this support typically involves activities on only the local hosts, *i.e.,* network and remote host characteristics and conditions are typically not considered during the configuration process.

An even more dynamic approach permits reconfiguration during the data-transfer phase, thereby accommodating dynamic changes in application requirements, transport system resources, and network characteristics. Very few transport systems provide support for adaptive reconfiguration, and those that do are experimental research projects [6, 19].

Given the increasing diversity of application requirements and network characteristics (combined with the fact that both may change dynamically), it appears that dynamically configured transport systems may support a wider range of application/network pairings more effectively than statically configured systems. Therefore, we are designing ADAPTIVE's architecture to support transport systems that execute lightweight and adaptive communication protocols that are configured flexibly.

**(C) No Explicit Support for Multimedia Applications and High-Performance Networks:** Many popular transport systems were designed and implemented before the new generation of multimedia applications and high-performance networks became widely available. For example, the BSD 4.3 UNIX TCP/IP implementation was originally designed for traditional data applications that required 100 percent error-free transmission over low-bandwidth, high-error network links such as ARPANET.

Therefore, TCP/IP (and in many cases the ISO TP[0-4] transport protocols) may prove inadequate for modern multimedia applications, since they do not incorporate (1) efficient control formats,[2] (2) fast connection establishment, (3)

[2]For example, neither TCP nor TP4 place their checksum in the packet trailer, thereby precluding simultaneous transmission and checksum computation [20]. Moreover, many fields in the TCP header are not word-aligned

multicast support, (4) security, (5) synchronized multimedia streams or (6) quality-of-service parameters such as prioritized real-time guarantees involving bounded delay/jitter and application-selectable levels of loss-tolerance [1, 21]. Moreover, most existing transport systems do not export multimedia services like isochronous and synchronous delivery guarantees from the underlying network to the application.

In addition, standard protocol suites were designed for low-speed, unreliable, congestion-prone datagram networks, rather than high-speed, congestion-controlled virtual-circuit networks such as ATM [22]. For example, the TCP/IP suite does not provide (1) explicit access control (although TCP's *slow start* and *multiplicative decrease* algorithms are used to simulate access control), (2) rate control and selective retransmission to handle congestion, or (3) long-delay link support such as large flow-control windows, non-wrapping sequence numbers, and precise round-trip timer calculations. Although extensions to TCP and IP have been proposed that address these limitations [23], these proposals are not yet been integrated into the base standards requirements.

### 2.3 Related Work

A growing number of projects address high-performance transport systems that are constructed out of flexible components that support lightweight and adaptive protocols. In addition to the research described in this section, other projects focusing on various aspects of flexible, lightweight, and adaptive transport systems and protocols include [3, 18, 20, 21, 24, 25, 26, 27]. The ADAPTIVE system is influenced by the Programmable Network Prototyping System (PNPS) [28], the *x*-kernel/Avoca projects [2, 7], the Function-based Communication SubSystem (F-CSS) [6], and the Multi-Stream Protocol (MSP) [19]. 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- and higher-layer protocols.

The *x*-kernel is a communications-oriented environment that supports protocol development and experimentation. It provides a "protocol backplane" consisting of uniform interfaces for reusable, "medium-granularity" communication service components such as message handling, multiplexing and demultiplexing, and event management. Protocol interrelationships are described and implemented via modular, highly-layered protocol graphs composed from virtual- and micro-protocols. Whereas the *x*-kernel focuses on OS support for medium-grain protocol activities, ADAPTIVE supports finer-grain protocol session functionality such as connection management, reliability management, and end-to-end transmission control.

Avoca uses the *x*-kernel as a run-time environment to support middle-layer protocol implementation and experimentation. Its primary emphasis is on flexible implementations of protocols like RPC, UDP, and TCP, which support traditional data applications running on traditional networks. ADAPTIVE, on the other hand, focuses on flexible and adaptive transport systems that support multimedia applications running on high-performance networks.

The Function-based Communication SubSystem (F-CSS) is a transport system architecture that supports dynamic reconfiguration of protocol stacks based on explicitly-specified application requirements. F-CSS utilizes a special-purpose environment based upon transputers, whereas ADAPTIVE resides in a general-purpose OS.

The Multi-Stream Protocol (MSP) is a "feature-rich" transport protocol designed to execute on parallel processors. In addition, MSP permits protocol configurations to change their mechanisms "on-the-fly" without loss of data (*e.g.,* switching the retransmission scheme from *go-back-n* to *selective repeat* within an active connection). Like the *x*-kernel, MSP focuses more 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 (as described in Section 3).

## 3 Overview of the ADAPTIVE System

ADAPTIVE is "A Dynamically Assembled Protocol Transformation, Integration, and Validation Environment." It is used to specify, configure, experiment with, analyze, and improve alternative transport system designs and implementations. In particular, we are developing ADAPTIVE to help ameliorate the inadequacies of existing systems described in Section 2.2 above. This section briefly describes the flexible and adaptive aspects of ADAPTIVE.

ADAPTIVE provides services that flexibly configure and reconfigure lightweight and adaptive transport system *sessions*. Its flexibility is derived from a repository of reusable mechanisms used to synthesize sessions that are customized to adapt to application requirements and network characteristics. These mechanisms include buffer and event managers, connection management, transmission control, and reliability management. A flexible transport system architectural design is essential for supporting prototyping, experimentation, and diversity.

Experience indicates that it is very difficult to specify one protocol that is optimal for all application/transport system/network combinations [29]. Therefore, instead of developing a single highly-complex, all-encompassing protocol, ADAPTIVE provides a transport system architecture that permits fine-grain selection and configuration of precisely specified protocol mechanisms [2].

ADAPTIVE also supports run-time adaptive reconfiguration that adapts to feedback from changes in applications requirements, transport system resources, and network characteristics. Adaptivity is important since applications and networks are dynamically changing entities, which are not necessarily served most effectively by static solutions. Section 4 describes the ADAPTIVE architecture in detail.

and the option formats are not fixed-sized, which increases header parsing overhead.

ADAPTIVE is distinguished by its flexible architecture that supports (A) application and network diversity and dynamism, (B) reduction in transport system overhead, (C) dual focus on both policies and mechanisms, and (D) feedback-guided monitoring and measurement. The following paragraphs discuss these points in detail:

**(A) Support for Both Multimedia Application Requirements and Network Characteristics:** ADAPTIVE configures transport system sessions based upon the network characteristics and the QoS requirements of applications. Other related work on transport systems typically emphasizes *either* diverse application requirements [6, 25, 30] *or* network characteristics [3, 19]. Moreover, existing implementations of transport systems tend to focus more on (1) traditional data applications such as bulk text file transfer and/or (2) traditional local area network environments such as Ethernet or Token Ring.

**(B) Reduction in Transport System Overhead:** A large body of research exists on both network protocols (see [31] for a survey) and the transport systems that support them (see [8] for our survey). Techniques for reducing transport system overhead involve various combinations of the following:

1. Utilize faster hardware for CPUs, busses, network controllers, and memory hierarchies.
2. Implement selected functions (*e.g.,* checksum calculations, message buffering, and demultiplexing) in special-purpose hardware.
3. Migrate some or all of the protocol processing activities to "off-board" processors [9, 20, 32] to reduce CPU interrupts and operating system context/process switching on the host computer.
4. Implement existing protocols more efficiently by (1) shortening the instruction paths executed for each packet and (2) improving the implementation of standard transport system support services such as message management and demultiplexing [10, 11, 33].
5. Design new lightweight and adaptive protocols that are tailored for high-speed, low error, and low delay network environments [19, 20, 34, 35, 36].
6. Develop alternative transport system architectures based on (a) vertical process architectures [7, 37], (b) parallel processing of protocol functions [3, 4, 19], (c) flexible protocol stacks that require fewer layers and/or are dynamically assembled [6, 25, 26, 30], and (d) modular and extensible transport system software that supports these flexible protocol stacks [2, 18, 38].

The ADAPTIVE architecture we are developing extends prior research on flexible transport system architectures. In particular, ADAPTIVE employs techniques from categories 4 (efficient implementations), 5 (lightweight and adaptive protocol designs), and 6 (alternative transport system architectures), with particular emphasis on flexible and modular software.

**(C) Focus on Both Policies and Mechanisms:** Related work on transport systems typically focuses on providing infrastructure *mechanisms* (*e.g.,* buffering, acknowledgment, retransmission, and timer schemes) that dictate *how* to configure and reconfigure the transport system. For example, [19] describes several adaptive mechanisms that support "on-the-fly" protocol reconfigurations. However, most existing research places less emphasis on the *policies* that determine *when* to adaptively reconfigure transport system mechanisms and *what* mechanisms the subsequent reconfigurations should contain. Conversely, related work that does address policies [6, 25, 30] is less specific with respect to the mechanisms that actually enforce these policies. For example, adequately supporting QoS requirements such as bounded delay and jitter involves complicated interactions between the network and the transport systems on both local and remote hosts.

The ADAPTIVE transport system architecture, on the other hand, specifically addresses both policies and mechanisms. The following two examples illustrate the distinction between policies and mechanisms:

- Transport system policies may switch a session's retransmission mechanism from *go-back-n* to *selective repeat* in the event that (1) the application's requirements change from multicast to unicast or (2) the congestion in the network increases beyond a specified threshold (resulting in greater packet loss due to queue overflows at intermediate switching nodes) [1, 3]. Note that it may be feasible to restore the *go-back-n* scheme when congestion subsides, thereby reducing buffering requirements at the receiver(s).
- The transport system may also contain policies that cause the reliability management mechanism to switch from "retransmission-based" to "forward error correction-based" when the round-trip delay time increases beyond some threshold (*e.g.,* when a route switches from a terrestrial link to a satellite link).

Note that in each of these cases, it is as important to understand the policies that dictate *when* to switch mechanisms and *what* to switch them to, as it is to know *how* to implement the mechanisms efficiently.

**(D) Support for Controlled, Empirical Experimentation via Performance Monitoring and Measurement:** It is difficult to empirically evaluate the advantages and disadvantages of different transport system designs without (1) a controlled implementation environment and (2) systematic methods for monitoring and measuring distributed application performance [10]. However, most transport systems are designed for performance rather than for experimentation. This makes it difficult to replace selected mechanisms (*e.g.,* acknowledgment or retransmission schemes) and measure the performance impact precisely. The ADAPTIVE protocol development process, on the other hand, explicitly employs measurement as part of its iterative, feedback-driven methodology, which consists of: (1) transport system session specification and configuration, (2) experimentation, (3) analysis of the results, and (4) using feedback from (3) to refine (1).

Monitoring and measuring the performance of different transport system configurations provides feedback on the consequences of selecting between alternative policies and

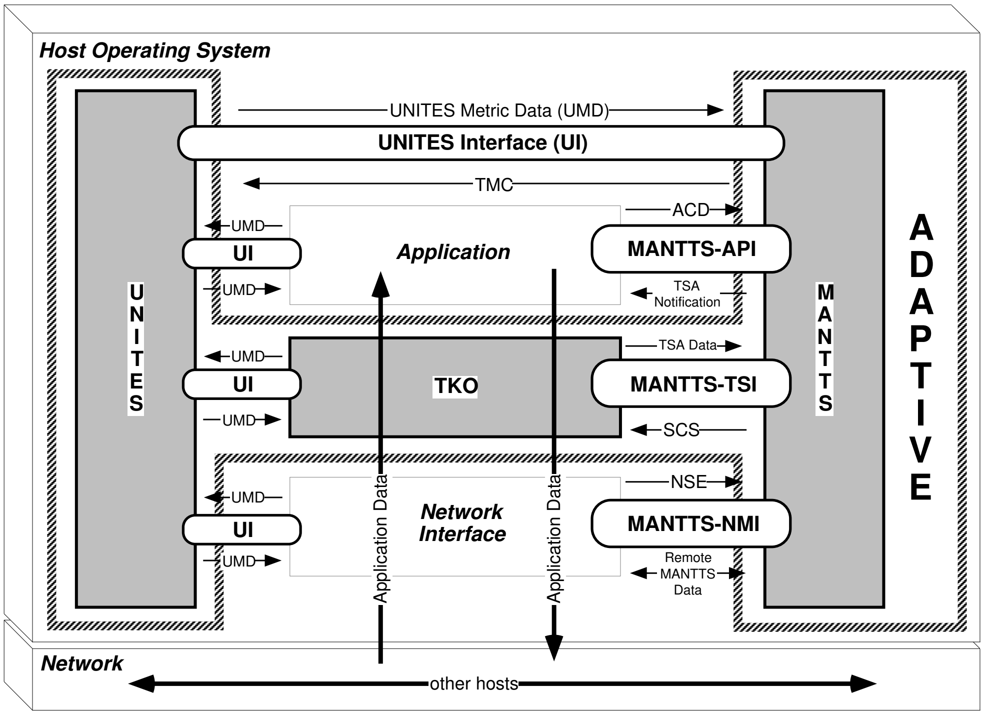


Figure 1: ADAPTIVE Architecture

mechanisms [28]. In addition, certain metric information (*e.g.,* packet loss and round-trip delay, CPU load, and memory utilization) may be used at run-time to determine *when* to reconfigure the transport system mechanisms. Other metrics for monitoring and measuring transport systems include throughput, response time, jitter, connection establishment time, and number of packet retransmissions [39]. Section 4.3 describes measurement issues in greater detail.

# 4 Design of the ADAPTIVE System

This section describes the architecture of the ADAPTIVE system and examines its major subsystems in detail. As shown by the shaded rectangles in Figure 1, ADAPTIVE's three main subsystems are:

• **MANTTS (***"Map Applications and Networks To Transport Systems"***):** MANTTS is responsible for choosing the appropriate set of policies and mechanisms to meet an application's quality-of-service (QoS) requirements (quality-of-service requirements involve both "qualitative" and "quantitative" factors). MANTTS negotiates with remote hosts and intermediate switching nodes, taking into account the dynamically changing network environment. It transforms the QoS requirements into a specification of the policies and mechanisms that comprise a transport system session configuration. Section 4.1 further describes MANTTS.

• **TKO (***"Transport Kernel Objects"***):** TKO synthesizes a customized lightweight transport system session configuration composed of reusable components selected from a *protocol mechanisms repository*. TKO instantiates these mechanisms to form an executable session object representation that guides the actions of an interpreter that performs protocol processing activities on PDUs flowing throughout the transport system. Section 4.2 further describes TKO.

• **UNITES (***"UNIform Transport Evaluation Subsystem"***):** UNITES supports network traffic monitoring, metric selection/collection/analysis/presentation, and performance measurement. In addition to enabling meaningful comparisons between different session configurations, UNITES provides feedback that assists MANTTS and TKO in determining when and how to dynamically alter certain mechanisms in a session. Section 4.3 further describes UNITES.

ADAPTIVE's architecture is decomposed into these three subsystems to facilitate the development of fine-grain, "plug-compatible" components that precisely meet the communication requirements of distributed applications. The following sections describe each ADAPTIVE subsystem in detail. Details of the interface to the native transport/operating system are excluded for clarity.

## 4.1 MANTTS

MANTTS ("Map Applications and Networks To Transport Systems") manages various resources (*e.g.,* message buffers, control blocks for open sessions, and available communication ports) and services (*e.g.,* protocol mechanisms providing session configuration and reconfiguration support) on ADAPTIVE host systems and intermediate switching nodes. For instance, MANTTS coordinates multiple related communication sessions (*e.g.,* determining the scheduling priorities of synchronized multimedia streams), guides the "requirement-driven" transformation process that synthesizes transport system session configurations, and monitors the network to detect and respond to dynamic changes in traffic conditions.

MANTTS is involved in several phases of communication including: (1) the connection negotiation and configuration phase (where it coordinates the initial session configuration), (2) the data transfer phase (where it coordinates the reconfiguration process), and (3) the connection termination phase (where it releases resources and recalculates transport system load information). The activities undertaken during each phase are discussed below.

#### 4.1.1 The Connection Negotiation and Configuration Phase

MANTTS is responsible for interactively negotiating the requested QoS requirements when an application initiates a connection. Negotiation occurs between the local and remote application and MANTTS entities, as well as the intermediate switching nodes. The negotiation phase determines the QoS that the transport system is capable of providing at the current time. Negotiation may be necessary if the QoS requested by the application is not available. However, negotiation need not determine an *optimal* configuration, as long as it produces one that meets the application's requirements (which may have a range of tolerance). If the negotiation is satisfactory to the participating applications and transport systems, the local and remote session configurations are instantiated (as described in Stage III below). If the negotiation/transformation is unsuccessful, on the other hand, (*e.g.*, due to resource limitations or network partitioning), there are two alternatives: (1) refuse the connection and (2) allow the application to re-negotiate at a lower quality of service.

In general, the negotiation process is based upon factors like the current network state and traffic volume, intermediate switching node queue lengths, host processing loads, and services available at the remote hosts. Local and remote MANTTS entities negotiate three basic categories: (1) parameters (*e.g.,* buffer space, initial window advertisements and scaling factors [40], segment size and maximum transmission unit (MTU) size, priorities for message delivery and scheduling, and timer settings for delayed acknowledgments and retransmissions), (2) mechanisms (*e.g.,* reliability management and transmission management mechanisms), and (3) representations (*e.g.,* fixed-size vs. variable-sized buffer management).

As shown in Figure 2, MANTTS configures a transport system session on behalf of the application via the following three-stage transformation process:

- **Stage I:** MANTTS transforms the QoS requirements into an appropriate set of policies known as a *Transport Service Class* (TSC) (TSCs are described further below).
- **Stage II:** MANTTS then transforms the TSC into a *Session Configuration Specification* (SCS). The SCS is a "blueprint" that specifies a set of protocol mechanisms that implement the selected TSC policies. The SCS is based upon information regarding static and dynamic network characteristics, along with information obtained from negotiating with remote application and MANTTS entities and intermediate switching nodes.
- **Stage III:** The mechanisms specified by the SCS are automatically synthesized by the *Transport Kernel Objects* (TKO) subsystem. TKO dynamically composes and instantiates customized session configurations. When executed on incoming and outgoing PDUs, these configurations are designed to deliver the requested communication service to applications.

The remainder of this section describes each stage of the transformation process in greater detail.

**Stage I – Transport Service Class Selection:** During the first transformation stage, MANTTS selects a *transport service class* (TSC) that corresponds to the application-specified communication requirements. As shown in Table 1, TSCs represent general classes of application requirements such as interactive isochronous traffic (*e.g.,* voice conversation), distributional isochronous traffic (*e.g.,* full-motion HDTV), non-isochronous real-time traffic (*e.g.,* robotics and manufacturing control), and non-isochronous, non-real-time traffic (*e.g.,* file transfer, remote login, and RPC) [6]. Each TSC embodies a set of related policy decisions that satisfy the application's QoS requests.

Applications may explicitly select a TSC to help simplify the subsequent transport system session configuration process. TSCs embody a set of default parameters, mechanisms, and/or representations, thereby requiring the negotiation of only a subset of these entities. Simplifying and reducing the configuration delay is important, since the benefits of a dynamically configured architecture are reduced if the configuration and/or reconfiguration process is overly time-consuming.

The MANTTS Application Programmatic Interface (MANTTS-API) provides the user interface for the ADAPTIVE system. For example, when an application initiates a connection it passes *application communication descriptor* (ACD) via the MANTTS-API. As shown in Table 2, ACDs specify several types of information including the remote session participant address(es), qualitative and quantitative quality-of-service parameters, and metric collection and reconfiguration requests. Multiple remote addresses are necessary if multicast service is requested. Participant addresses indicate certain characteristics of intermediate switching nodes and remote end systems such as available bandwidth, MTU, latency, and bit error rates. Knowledge of these characteristics enables MANTTS to determine more appropriate policies and mechanisms. Duration is an important quantitative QoS parameter for digital continuous media (DCM) sessions [41]. It is an essential configuration parameter for ADAPTIVE, since it is not generally useful to dynamically reconfigure sessions that have very low duration.

**Stage II – Session Configuration Specification:** During the second stage of configuration, MANTTS transforms the

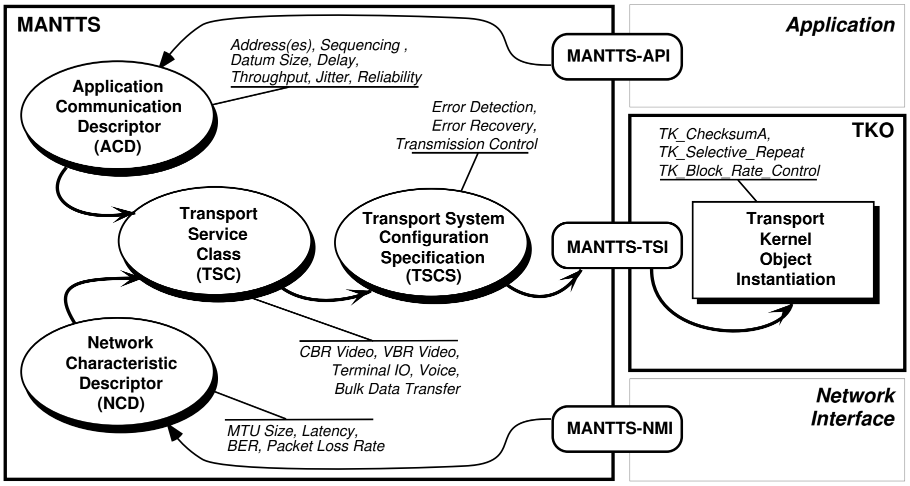


Figure 2: MANTTS Transformation Model

| Parameter Name | Description | Example Specifiers |
|---|---|---|
| Remote Session Participant Address(es) | Specifies $\geq$ addresses of remote end-systems that comprise the communication association. | |
| Quantitative QoS Parameters | Specifies the performance criteria requested by the application. | Peak and average throughput, minimum and maximum latency and jitter error-rate probabilities, duration, etc. |
| Qualitative QoS Parameters | Specifies the functionality of behavior requested by the application. | Sequenced/non-sequenced delivery, duplicate sensitivity, explicit/implicit connection management, (byte/packet/block)-based transmission and acknowledgment. |
| Transport Service Adjustment (TSA) | Actions to perform when changes occur in local or remote hosts or the network. | <*condition, action*> pairs that indicate the actions to perform when certain conditions are true. |
| Transport Measurement Component (TMC) | Specifies performance metrics to collect for this particular communication session. | Parameter selection, sampling rate, presentation format, etc. |

Table 2: The ADAPTIVE Communication Descriptor Format

selected TSC into a corresponding session configuration specification (SCS). The resulting SCS represents a set of protocol mechanisms that implement the policies embodied in the TSC. MANTTS produces the SCS by reconciling the TSC with the network characteristics associated with the indicated remote addresses. A *network state descriptor* maintained by the MANTTS Network Monitor Interface (MANTTS-NMI) samples, records, and estimates the current state of dynamic network characteristics.

The initial configuration of a transport system session in an ADAPTIVE host may choose between one of two alternatives:

**1. Implicit Negotiation:** MANTTS supports implicit connection management, where configuration information is piggybacked along with the application's first PDU. This is useful for latency-sensitive applications (*e.g.,* "request-response"-style network file servers) that must not incur any QoS negotiation delay. To support implicit negotiation, remote MANTTS entities must supply reasonable values for default configurations.

Longer-duration applications may also choose implicit negotiation, since eliminating the multiple round-trip "handshake" overhead is beneficial for sessions running over long-delay links [1]. If necessary, it is possible to reconfigure an active session dynamically at a later point, using the renegotiation mechanisms described in Section 4.1.2 below.

**2. Explicit Negotiation:** When an application requests either explicit connection management and/or peer-to-peer negotiation, the initiating local MANTTS entity uses an out-of-band signaling channel to exchange and negotiate configuration parameters with remote MANTTS entities. Out-of-band signaling helps to optimize the main data transfer path, since this path does not interpret packets containing control information [31].

Figure 3 illustrates the separate control and data paths used by signaling and data units during the negotiation process and subsequent data transfer. The additional time spent negotiating QoS should improve the overall performance for longer-duration, high-bandwidth connections, since the resulting configurations more accurately reflect the application requirements and network characteristics [31]. Moreover, the negotiation process may be combined with explicit connection management that occurs during the initial 2-way or 3-way handshake. This allows local configuration activities to proceed in parallel with remote negotiation activities.

**Stage III – Transport System Synthesis:** In the third stage of configuration, MANTTS submits the session configuration specification (SCS) to the *Transport Kernel Object* (TKO) subsystem via the MANTTS Transport System Interface (MANTTS-TSI). TKO transforms the SCS into a *transport system session configuration*, which is customized for the particular applications and networks involved. This

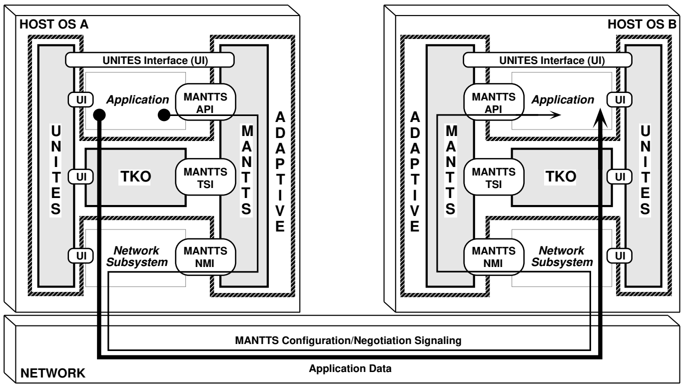


Figure 3: Connection Configuration

final transformation stage involves (1) synthesizing the specified protocol mechanisms (*e.g.,* buffer management, transmission control, and reliability management) from the TKO class repository, (2) instantiating an executable representation that is used by a protocol interpreter to invoke operations on PDUs in the appropriate order, and (3) instrumenting the dynamically synthesized mechanisms using the UNITES measurement facilities to satisfy the application's metric collection requests. Instrumentation is based on both application requests and global UNITES metric collection requests. Section 4.2 describes the TKO subsystem in detail; Section 4.3 describes the UNITES subsystem in detail.

#### 4.1.2 Data Transfer and Reconfiguration Phase

The ADAPTIVE architecture adapts to dynamic changes in application requirements and network characteristics by explicitly and/or implicitly reconfiguring transport system sessions. The reconfiguration process is similar to the initial configuration process, and may involve decreasing or increasing the QoS.

Explicit reconfiguration is initiated by a local or remote application request such as changing from unicast to multicast communication or requesting a higher or lower quality of service. Remote MANTTS entities are notified if a new SCS must be generated to satisfy the request, and the corresponding transport system session configurations are updated. Explicit reconfiguration requests are sent via the same out-of-band signaling channel used for connection negotiation.

MANTTS automatically detects situations where implicit reconfiguration is necessary. These situations typically occur when parameters monitored by MANTTS exceed thresholds specified by transport system policies. For example, local or remote MANTTS entities may initiate renegotiation in response to increases or decreases in buffer space, round-trip delay, or changes in processor load. Similarly, changes in network conditions may be detected by the MANTTS Network Monitor. These conditions include intermediate switching node failure, lower throughput due to increased network congestion, or routing changes. For example, if a long-duration connection switches to a high-delay path (*e.g.,* if intermediate node failures cause routes to change from a terrestrial link to a satellite link), it may become necessary to reconfigure certain mechanisms such as the retransmission scheme or window scaling options.

During reconfiguration, one of the following actions occurs:

- **Adjust the TSC:** change the Transport Service Class to provide a QoS that is more suitable to the new conditions (*e.g.,* benefitting an application that has changed video coding schemes and now requires isochronous service). This change potentially generates a completely new or partially modified SCS.

- **Adjust the SCS:** replace or adjust one or more parameters or mechanisms used by the transport system to compensate for the indicated change (*e.g.,* increase the inter-PDU gap used by the rate control mechanism in response to perceived network congestion). In this case, no change occurs in the corresponding TSC.

- **Application-Specific:** notify the application when changes occur in operating conditions (*e.g.,* a reduction in receiver's buffer space) via a *call-back* mechanism that allows it to react appropriately (*e.g.,* begin transmitting data using an application-specific compression or component coding scheme).

#### 4.1.3 Connection Termination Phase

There are several different types of connection termination semantics supported by ADAPTIVE, including graceful (*i.e.,* no loss of buffered data) and non-graceful (*i.e.,* potential loss of buffered data). In addition to shutting down the communication channel, the termination phase provides an opportunity for MANTTS to release allocated resources and recalculate transport system load information. Both explicit and implicit connection termination is supported.

### 4.2 TKO

The TKO ("Transport Kernel Objects") subsystem provides a modular and extensible framework for configuring and reconfiguring transport systems sessions. Modularity enhances transport system flexibility by providing uniform interfaces for "plug-compatible" protocol and session processing components. Extensibility enhances transport system adaptivity by permitting the addition of new and/or alternative services at run-time. Flexibility and adaptivity, in turn, help support the diversity and dynamism of applications and networks.

As shown in Figure 4, the TKO subsystem consists of the two distinct levels of abstraction described below: (1) TKO *protocol architecture* and (2) TKO *session architecture*. Each level performs a well-defined set of services for the transport system. TKO is implemented in C++ since (1) C++ integrates context information together with operations to support object-oriented design and programming and (2) C++ directly supports flexibility and adaptivity via inheritance and dynamic binding. Inheritance is a particularly use-

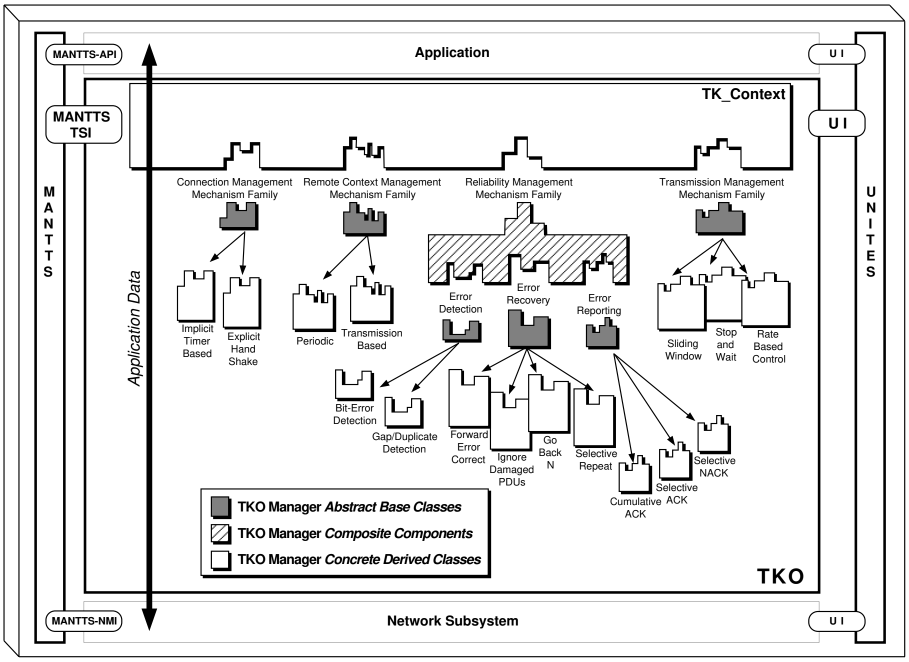


Figure 4: TKO Protocol and Session Architectures

ful technique for maximizing sharing, reuse, and extensibility of TKO components.

### 4.2.1 TKO OS Protocol Architecture:

The TKO protocol architecture is a repository of "medium-granularity" C++ classes that insulate the transport system from the underlying operating system environment. The ADAPTIVE TKO protocol architecture design is influenced by the *x*-kernel [7] and Conduit [18] communication systems. The TKO protocol architecture classes provide uniform interfaces for accessing protocol support services such as timers, message buffering, and protocol graph operations that insert, delete, and/or alter protocol objects. There are four fundamental TKO protocol architecture classes: `TKO_Event`, `TKO_Message`, `TKO_Protocol` and `TKO_Session`:

• `TKO_Event`: Many protocols must respond to temporal events such as retransmission timer expiration or periodic update requests [42]. The `TKO_Event` class provides event management operations such as `TKO_Event::schedule`, `TKO_Event::expire`, and `TKO_Event::cancel`. `TKO_Event` objects schedule themselves to expire one or more times (*i.e.,* they are "one-shot" or periodic), they may be cancelled, and they are triggered to expire asynchronously by the operating system's timer facility.

• `TKO_Message`: Performance measurements indicate that memory-to-memory copying is a significant source of transport system overhead [43]. Therefore, some form of buffer management is required to avoid unnecessary copying when (1) moving messages between protocol layers and (2) when adding or deleting message headers and trailers. The `TKO_Message` class provides uniform interfaces for services that create, copy, prepend, and split messages. `TKO_Message` objects are logically divided into two distinct regions: the *header* and the *data*. The data region efficiently supports "lazy copying" operations, as well as handling message fragmentation and reassembly. Likewise, the header region allows operations that efficiently prepend header information onto a message (`TKO_Message::push`) and later strip it off (`TKO_Message::pop`).

• `TKO_Protocol`: Conventional protocol suites are specified as a hierarchy of layers, where each layer performs well-defined services for layers above it, while making use of services from layers below it. The `TKO_Protocol` class provides uniform interfaces for protocol objects used to compose protocol suites like TCP/IP, OSI, and XNS. The service interface for `TKO_Protocol` objects provides (1) management operations for manipulating "protocol graphs" (which express the relationships between various protocol objects) and (2) operations that create, destroy, and demultiplex and multiplex messages to `TKO_Session` objects. Each `TKO_Protocol` object contains 0 or more `TKO_Session` objects.

• `TKO_Session`: The `TKO_Session` class forms the junction between the TKO protocol architecture and TKO session architecture services (as shown in Figure 4). `TKO_Session` objects encapsulate certain context information (*e.g.,* local and remote addresses) needed to correctly and efficiently process a session's incoming and outgoing messages. Associated with this context information are operations used for (1) sending and receiving `TKO_Message` objects that flow through `TKO_Session` objects, (2) dynamic session attachment (*i.e.,* allocating and linking together new sessions to form session graphs), and (3) dispatching system calls that store and/or retrieve session control information (*e.g.,* determining host and peer network addresses or determining the MTU for a given network interface).

These four fundamental classes provide the infrastructure that supports the TKO session architecture services described below.

### 4.2.2 TKO Session Architecture

Like the TKO protocol architecture, the TKO session architecture is designed as a repository of reusable C++ transport system mechanisms. However, each component in the TKO session architecture repository encapsulates "finer-grain" mechanisms associated with session activities such as connection management, transmission control, and reliability management. The remainder of this section describes the organization of the primary TKO session architecture components, explains how transport system sessions are synthesized and discusses several performance optimization techniques.

To support flexible configuration and reconfiguration, TKO session architecture is organized as a collection of C++ inheritance hierarchies. As shown in Figure 5, these inheritance hierarchies are "rooted" at *abstract base classes* that provide uniform interfaces for session activities. Specialized instances of these mechanisms are derived from an appropriate abstract base class and combined to form complete session configurations. Two `TKO_SA` components, `TKO_Context` and `TKO_Synthesizer`, manage and maintain these mechanisms as described below:

• **TKO_Context:** Figure 4 depicts how each `TKO_Session` object points to one or more `TKO_Context` objects that are customized for the associated application's QoS requirements. As shown in Figure 5, `TKO_Context` objects possess several different types of components that (1) provide functions to perform certain session activities (*e.g.,* connection management, reliability management, and transmission management) and (2) maintain context information (*e.g.,* round-trip time estimates, sequence numbers, or flow control window advertisements) necessary to support these functions. This encapsulation of context and functions facilitates rapid prototyping and incremental protocol development.

Each `TKO_Context` object contains a table of pointers to C++ abstract base classes that define the session's behavior. Table entries point to objects that are either *concrete derived subclasses* or *composite components*. Complete session objects are synthesized by *composing* and *instantiating* concrete derived subclasses and composite components that are appropriate for the specified application requirements and network characteristics.

Concrete derived subclasses are formed by selectively inheriting certain context information and functions from existing abstract base classes. These subclasses "specialize" basic session mechanisms (*e.g.,* `Sliding_Window` is derived from the abstract base class `Transmission_Management`). Likewise, composite components manage objects of several related subcomponents that interoperate in well-defined ways. For example, the `Reliability_Management` composite component in Figure 5 performs error detection, error reporting, and error recovery. Composite components are efficiently replaced and recombined in their entirety, thereby minimizing run-time overhead.

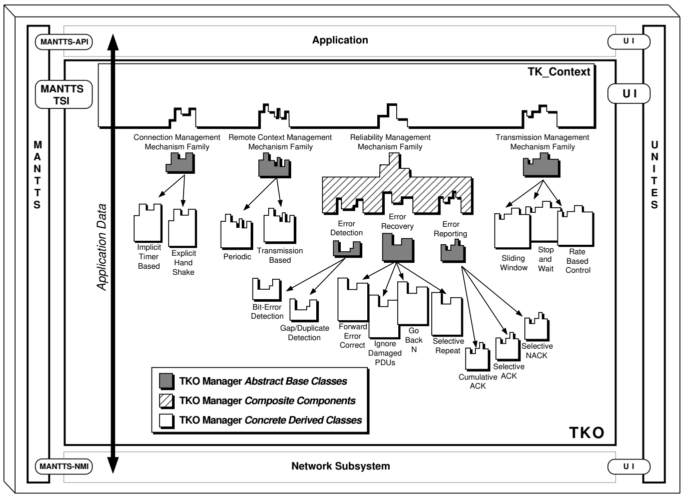


Figure 5: TKO Context

• **TKO_Synthesizer:** The `TKO_Synthesizer` is responsible for overseeing the configuration and reconfiguration of `TKO_Context` objects. As shown in Figure 5, the synthesizer receives the session configuration specification (SCS) from the MANTTS-TSI and transforms it into an efficient, lightweight `TKO_Context` session instantiation. A `TKO_Synthesizer` is also responsible for instrumenting and monitoring the transport system session resources specified in Transport Measurement Component.

The `TKO_Synthesizer` permits fine-grain, per-session control over its associated `TKO_Context` bindings by instantiating entries in the `TKO_Context` table at either configuration-time or run-time. In the latter case, the `TKO_Synthesizer` coordinates dynamic session reconfiguration by re-instantiating the updated session mechanism(s). For instance, each class object provides a means for replacing itself with a different class object via the `segue` mechanism supplied in all abstract base classes. `Segue` supports reconfiguration by permitting certain class object bindings to change dynamically. This approach differs from the BSD 4.3 UNIX network architecture, where protocol operators are statically bound to session control blocks through pointers to C functions. In the BSD scheme, all sessions associated with a particular protocol object use the same bindings, which are fixed at link-time and do not change.

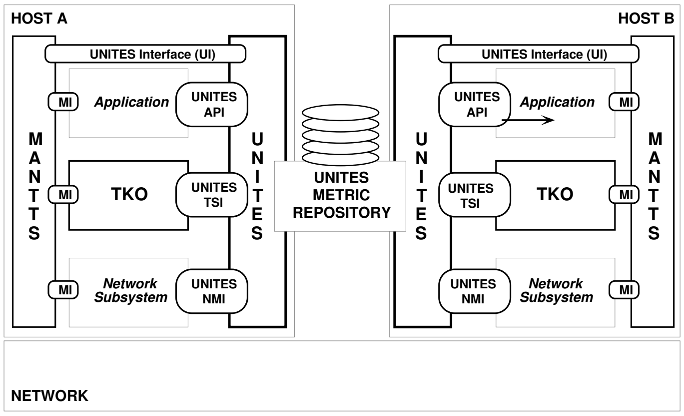


Figure 6: UNITES Architecture

TKO session architecture flexibility is supported by the C++ "virtual function" dynamic binding mechanism [44]. Although dynamic binding enhances *flexibility* (and thereby facilitates *prototyping* and *experimentation*), it increases processing overhead somewhat due to the extra level of indirection required to dispatch C++ virtual functions. To reduce this overhead, TKO employs a technique known as *customization* [45], which generates non-dynamically bound configurations for circumstances where performance is more important than flexibility. Customization incurs a time-space tradeoff, however, since inline expansion of many customized kernel objects may lead to excessive "code bloat" (see [46] for a discussion of a similar problem with the Synthesis Kernel).

In addition, the TKO session architecture maintains a cache of customized "`TKO_Templates`" that further optimize the instantiation process. `TKO_Templates` provide default transport system session configurations for commonly requested SCSs. These templates reduce the complexity and duration of the connection negotiation phase. There are two general types of `TKO_Templates`:

1. *Static Templates* are guaranteed not to change. This allows more efficient implementation, since the mechanisms may be completely customized (*e.g.,* inline expanded rather than dynamically dispatched). Static templates are also used to implement backward compatibility with existing protocols like TCP.

2. *Reconfigurable Templates* may change at some point during the communication session. Although this approach is generally more flexible, it is also less efficient since additional indirection is required (though not as much as is needed by a dynamically synthesized configuration).

If a pre-assembled `TKO_Template` does not exist to match an SCS request, TKO session architecture is responsible for dynamically synthesizing one [46].

## 4.3 UNITES

The UNITES ("Uniform Transport and Evaluation Subsystem") subsystem facilitates transport system protocol experimentation by coordinating *metric specification*, *metric collection*, *metric analysis*, and *metric presentation*. A fundamental goal of ADAPTIVE is to provide a framework that supports controlled hypothesis testing of different transport system session configurations. For example, we are currently experimenting with alternative protocol design configurations to determine those that best serve particular pairings of multimedia application requirements and underlying network characteristics. Information collected by the UNITES metrics quantifies trade-offs and interactions among different configurations, thereby providing meaningful design and implementation evaluations.

UNITES metrics are divided into two classes: *blackbox* and *whitebox*. These classes differ depending on whether or not the metric collection mechanisms require internal instrumentation of transport system session configurations. Both blackbox and whitebox metrics assist the tuning of transport system configurations by quantifying the performance consequences of certain design and implementations decisions.

Blackbox metrics are collected without knowledge of internal implementation details. They include throughput (defined as the number of packets transmitted per second) and latency (defined as the packet round-trip time for interactive traffic). Whitebox metrics, on the other hand, require internal instrumentation of the session configurations. These metrics include connection establishment and termination latency, number of packet (re)transmissions, the number of instructions required to execute a protocol function, interrupt and scheduling overhead, the degree of jitter (defined as the variance in
the delay), and packet loss. Collecting white-box metrics is very difficult without a development and testing environment like ADAPTIVE that supports controlled experimentation.

As illustrated in Figure 6, the UNITES Metric Repository stores the collected metric information in a database. A repository is necessary when many active connections are instrumented and monitored, since too much data is generated to collect and process in real-time [28]. Users can access this information in various ways, including interactive graphic displays or standard network management protocols such as SNMP or CMIP. This metric data is presented in either a systemwide, per-host, or per-connection manner. UNITES monitors transport system sessions in two primary ways:

1. Application programs use the Transport Measurement Component parameter in the ADAPTIVE Communication Descriptor to indicate metrics they want UNITES to collect. The TKO subsystem then selectively instruments the synthesized configurations and the metrics are automatically collected by UNITES at run-time.

2. To support experimentation, metrics also may be requested using either a graphics-based or language-based interface. Sjodin *et al.*'s work is an example of a language-based approach [39]. They define a specification language that indicates what measurements to

collect and what traffic to generate. Figure 6 shows a graphical interface for specifying certain metrics to collect on a per-host basis.

# 5 Summary

Adequately supporting next-generation multimedia applications necessitates transport systems that tailor their services to precisely meet the communication requirements of both applications and high-performance networks. ADAPTIVE is a flexible and adaptive transport system architecture designed to provide these services.

ADAPTIVE provides (1) support for application and network diversity and dynamism, (2) reduction in transport system overhead, (3) focus on both policies and mechanisms, and (4) feedback-guided monitoring and measurement. In particular, ADAPTIVE's flexible software architecture facilitates an "experimentation-based" protocol development methodology. ADAPTIVE enables precise measurement of application and network performance changes that result from selectively modifying certain transport system mechanisms (*e.g.,* measuring the effect of switching from *implicit* to *explicit* connection management or from *selective repeat* to *go-back-n* retransmission).

We are currently designing and implementing a prototype implementation of ADAPTIVE. This prototype is written in C++, and is hosted on both the *x*-kernel and System V release 4 STREAMS. We are using the prototype to experiment with different transport system configurations that support multimedia applications (*e.g.,* network voice and video) running on several different networks (*e.g.,* Ethernet, Tree Network [47], DQDB, and FDDI).

# Acknowledgments

We would like to thank Hung Huang, Unmesh Rathi, and Girish Kotmire for their help in designing the ADAPTIVE system and for proof-reading earlier drafts of this paper.

# References

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

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

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

[4] M. Zitterbart, "High-Speed Protocol Implementations Based on a Multiprocessor-Architecture," in *Proceedings of the 1st International Workshop on High-Speed Networks*, pp. 151–163, May 1989.

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

[6] M. Zitterbart, B. Stiller, and A. Tantawy, "A Model for Flexible High-Performance Communication Subsystems," Tech. Rep. TC 17801 (78023), IBM Research Division, February 1992.

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

[8] 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.

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

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

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

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

[13] D. R. Cheriton, "The V Distributed System," *Communications of the ACM*, vol. 31, March 1988.

[14] R. V. Renesse, H. V. Staveren, and A. S. Tanenbaum, "Performance of the Amoeba Distributed Operating System," *Software – Practice and Experience*, vol. 19, pp. 223–234, March 1989.

[15] M. N. Group, "Network Server Design," Tech. Rep. CMS-CS-89-31, Carnegie Mellon University, August 1989.

[16] B. B. Welch, "The Sprite Remote Procedure Call System," Tech. Rep. UCB/CSD 86/302, Computer Science Division (EECS), University of California, Berkeley, CA 94720, June 1986.

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

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

[19] 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.

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

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

[22] D. F. Box, D. C. Schmidt, and T. Suda, "Alternative Approaches to ATM/Internet Interoperation," (Tucson, Arizona), pp. 1–5, IEEE, 1992.

[23] V. Jacobson, R. Braden, and L. Zhang, "TCP Extensions for High-Speed Paths," *Network Information Center RFC 1185*, pp. 1–21, October 1990.

[24] G. Blair, G. Coulson, F. Garcia, D. Hutchinson, and D. Shepherd, "Towards New Transport Services to Support Distributed Multimedia Applications," in *4th IEEE ComSoc International Workshop on Multimedia Communications*, (Monterey, California), pp. 250–259, IEEE, 1992.

[25] T. Plagemann, B. Plattner, M. Vogt, and T. Walter, "A Model for Dynamic Configuration of Light-Weight Protocols," in *IEEE Third Workshop on Future Trends of Distributed Systems*, pp. 1–9, IEEE, 1992.

[26] B. Heinrichs, "Versatile Protocol Processing for Multimedia Communications," in *4th IEEE ComSoc International Workshop on Multimedia Communications*, (Monterey, California), pp. 160–169, IEEE, 1992.

[27] G. Parulkar and J. Turner, "Towards a Framework for High Speed Communication in a Heterogeneous Networking Environment," *IEEE Network*, March 1990.

[28] 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.

[29] L. Peterson and S. O'Malley, "TCP Extensions Considered Harmful," *Network Information Center RFC 1263*, pp. 1–19, 1991.

[30] C. Tschudin, "Flexible Protocol Stacks," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Zurich Switzerland), pp. 197–205, ACM, Sept. 1991.

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

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

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

[34] 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.

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

[36] D. R. Cheriton and C. L. Williamson, "VMTP as the Transport Layer for High-Performance Distributed Systems," *IEEE Communications Magazine*, vol. 27, pp. 37–44, June 1989.

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

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

[39] P. Gunningberg, M. Bjorkman, E. Nordmark, S. Pink, P. Sjodin, and J.-E. Stromquist, "Application Protocols and Performance Benchmarks," *IEEE Communications Magazine*, vol. 27, pp. 30–36, June 1989.

[40] V. Jacobson and R. Braden, "TCP Extensions for Long-Delay Paths," *Network Information Center RFC 1072*, pp. 1–16, Oct. 1988.

[41] J. C. Pasquale, G. C. Polyzos, E. W. Anderson, K. R. Fall, J. S. Kay, V. P. Kompella, S. R. McMullan, and D. Ranganathan, "Network and Operating System Support for Multimedia Applications," Tech. Rep. CS-91-186, University of California, San Diego, 1990.

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

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

[44] Bjarne Stroustrup and Margret Ellis, *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[45] C. Chambers and D. Ungar, "Customization: optimizing compiler technology for SELF (a dynamically-typed object-oriented language)," in *Proc. SIGPLAN*, ACM, 1989.

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

[47] H. K. Huang, T. Suda, and Y. Noguchi, "LAN With Collision Avoidance: Switch Implementation and Simulation Study," in *Proceedings of the 15th Conference on Local Computer Networks*, 1990.