ADAPTIVE

An Object-Oriented Framework for Flexible and Adaptive Communication Protocols

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

Abstract

Traditional transport systems do not adequately provide the functionality or flexibility required by existing and future multimedia applications. Conventional protocol architectures based on a static configuration of relatively few protocols are incapable of providing the level of performance the channel is capable of producing while still performing the processing needed by the application. Multimedia applications require transport systems that can be configured to match the the functional requirements of diverse multimedia traffic sources as well as capable of adapting to the dynamism inherent in multimedia applications and heterogeneous internetworks.

This paper describes ADAPTIVE, a transport system architecture to support multimedia applications for high-speed networks. The ADAPTIVE system applies object-oriented design and implementation techniques to build an integrated framework for protocol specification, composition, prototyping and experimentation. It utilizes a hierarchical specification technique that allows both the policies of a communication session to be specified and the actual mechanisms used to carry out these policies. Its monitoring and analysis facilities provide a rich environment for controlled experimentation through the use of rapid prototyping and integrated instrumentation.

1 Introduction

Traditional transport systems do not adequately provide the functionality or flexibility needed by existing and future multimedia applications and high speed networks. Applications currently must either (1) accept wholesale the functionality and behavior of an available protocol (*e.g.*, TCP, UDP, TP4, or VMTP), or (2) attempt to provide its own transport subsystem by building on lower level communication service primitives. Due to the diversity of application requirements and the paucity of available protocols on most systems, approach (1) often leads to lowest-common-denominator solutions that actually only satisfy the requirements of a narrow range of their target applications. Approach (2) leads to multiple, ad hoc, implementations that are not easily extended, modified, or shared. This also places the burden of protocol processing on the application programmer, who may not be fluent in the design and implementation of communication protocols. To alleviate this situation, future transport systems must provide communication service that is flexible and adaptive to (1) application diversity, (2) network diversity, and (3) host system diversity.

Application Diversity: Distributed multimedia applications impose unique performance constraints on the underlying communication medium and the supporting transport system that are more demanding and dynamic than those previously encountered in traditional data applications. The presence of these applications increases the dynamism of the underlying network and their supporting transport systems due to the high degree of variance in traffic characteristics exhibited by the applications' data sources (*e.g.*, highly bursty, high bandwidth variable bit rate video sources, relatively steady, low bandwidth digitized voice sources, short transactional-based sources). Transport systems providing communication service that utilizes traditional communication protocol suites typically offer very few options with respect to both the *quality* of service (*e.g.*, high throughput, low delay) and the *functionality* of service (*e.g.*, reliable in-order data stream, best-effort datagram) provided. Existing and future multimedia applications require various levels of performance (*e.g.*, peak/average bandwidth, maximum delay, low jitter) and behavior (*e.g.*, synchronization, network-kernel-application delivery, error correction), that are not adequately addressed in existing systems.

Network Diversity: The diversity of network characteristics encountered by distributed multimedia applications are due to (1) the heterogeneity of internetworking environments, (2) dynamic or multipath routing, and (3) fluctuations in network state caused by the traffic sources described above. Network characteristics that vary across network environments include *channel speed* (*e.g.*, 10Mbps for Ethernet, 100Mbps for FDDI, 155Mbps or 622Mbps for ATM), *maximum data transfer unit* (*e.g.*, 1500 octets for Ethernet, 9188 octets for SMDS, 48 octets for ATM), *available service types* (*e.g.*, datagram, virtual circuit, multicast, broad-

[1] This paper appeared in the proceedings of the fourth IFIP conference on High Performance Networking in Liege, Belgium, December 1992.

cast) and *access control scheme* (*e.g.*, CSMA/CD, token-passing, switch-based). Network characteristics that may vary dynamically over the lifetime of an association include the aforementioned characteristics, which may fluctuate either due to a change in routing, the use of multipath routing, or a change in the number or location of participants in a communication session. Additionally, network characteristics such as packet loss rate and delay can vary greatly over the lifetime of an association due to transient network congestion.

Host System Diversity: A large degree of diversity exists in the degree and nature of support provided by host systems for communication protocols. This diversity appears in both the available hardware (*e.g.*, CPU, network interface, memory hierarchy) and the supporting system software. Software-related issues such as scheduling mechanisms (*e.g.*, *x*-kernel[1] lightweight processes vs. STREAMS[2] `service` routines), user-kernel data delivery mechanisms (*e.g.*, BSD socket layer [3] vs. *x*-kernel upcall mechanism), buffer management schemes (*e.g.*, BSD `mbufs` vs. STREAMS `mblk_ts`) and protocol composition and demultiplexing mechanisms (*e.g.*, STREAMS modules vs. *x*-kernel protocol and session objects). Hardware issues that are variable across host systems include processor architectures (*e.g.*, uniprocessor vs. shared memory multiprocessor [4] vs. message passing multiprocessor[5, 6]), explicit support for protocol processing (*e.g.*, performing all protocol processing off-board processors[7, 8], specialized hardware to assist a single protocol function [9]), effects of interrupts on overall system performance (*e.g.*, number of interrupts required to move data between the host system memory and the network interface, performance penalty from interrupt-driven processing due to the amount of context a processor must save across interrupts, cache invalidation and pipeline flushing).

The ADAPTIVE System: ADAPTIVE is "A Dynamically Assembled Protocol Transformation, Intergration, and Validation Environment." The ADAPTIVE system [10] has been designed to address the diversity described above by providing: (1) a flexible and adaptive kernel of protocol mechanisms that provide a framework for protocol composition, (2) a unified scheme for specifying both the policies and the mechanisms that are used to provide communication services, and (3) an integrated environment for the specification, collection, and presentation of performance data. As shown in Figure 1, ADAPTIVE's three main subsystems are:

1. *Map Applications and Networks To Transport Systems (MANTTS)* – MANTTS interacts with the entities of a communication session to select the policies and mechanisms that will satisfy an application's communication requirements given the diverse needs of an application and the dynamic state of the network. Section 3 describes MANTTS in detail.

2. *Transport Kernel Objects (TKO)* – TKO instantiates precisely-tailored *transport system session contexts* from a library of reusable protocol mechanisms. These sessions maintain one or more *streams*, each of which corresponds to an independent unidirectional data stream between two logical endpoints of communication. Section 4 describes TKO in detail.

3. *UNIform Transport Evaluation Subsystem (UNITES)* – UNITES provides an infrastructure for traffic monitoring, performance evaluation and protocol instrumentation. Section 5 describes UNITES in detail.

2 ADAPTIVE Design Principles

Adequately supporting the diversity of application requirements and network characteristics described in Section 1 requires a flexible transport system architecture that provides communication service appropriate for the specific traffic sources and underlying network technologies [11, 12, 13]. ADAPTIVE allows the behavior of a communication session to be precisely tailored to the required service by implementing a protocol in terms of a set of independently recombinable protocol mechanisms. This independence is achieved through the strict use of uniform abstract interfaces to each set of functionally-similar mechanisms. Using this composition scheme facilitates the following:

2.1 Controlled Protocol Experimentation

By holding all other mechanisms within a session constant, the effects of choices within one subset of the mechanisms can be accurately observed without undue interaction with the rest of the protocol. For example, the effect on protocol performance due to changing the Connection Management function from one that is implicit timer-based to one that is explicit handshake-based can be attributed to the mechanism selection, as all other factors can be held constant. Previous comparisons of various protocol mechanisms [14, 15, 16, 17] have been done largely based on their implementations within the context of a complete protocol, thus making it difficult to isolate a single mechanism from its interactions with the rest of the system. More accurate conclusions may be reached as to the suitability of a given mechanism using controlled experimentation techniques.

2.2 Flexible Protocol Engineering

By leveraging off of established software engineering techniques, the task of *correctly* implementing a communication protocol can be made less complex than using traditional implementation methods. ADAPTIVE provides a framework of reusable mechanism *objects* [18] that allow protocols to be developed from new and existing component mechanisms that are independently implemented, tested, and maintained. ADAPTIVE implements protocol mechanisms

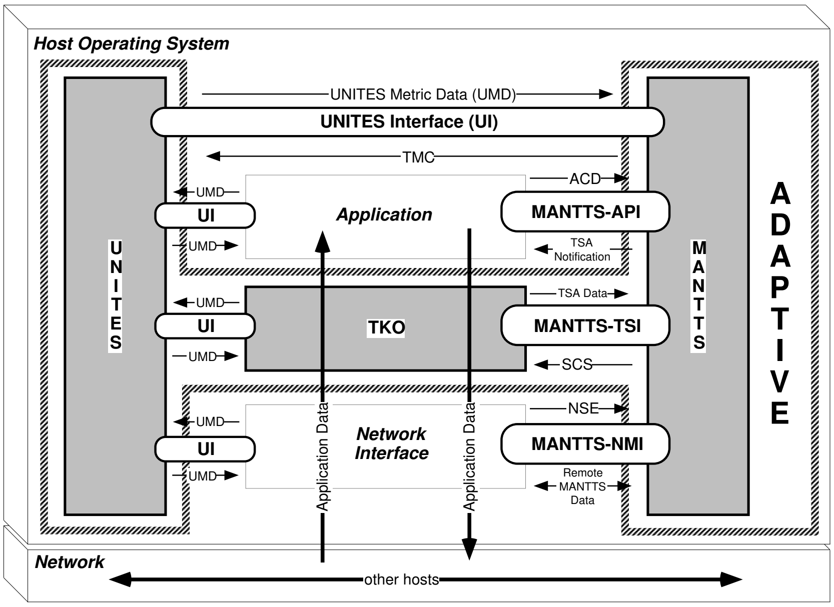

Figure 1: ADAPTIVE System Architecture

to comply with uniform abstract interfaces, allowing the one implementation of a protocol mechanism to be replaced by another (*e.g.*, *go-back-n* replacing *selective repeat* error recovery) written to the same interface without affecting the implementation of the other constituent mechanisms. Protocol mechanisms are implemented as C++ objects [19] that encapsulate both the current *state* of a protocol and the *operations* that are performed to implement the mechanism as one unified abstract data type. Instances of these data types are then instantiated and configured at run-time to provide the desired protocol function. Implementing protocol mechanisms as objects yields several desirable results:

1. *Information Hiding* – details specific to the internal implementation of a given mechanism are hidden behind a uniform interface [20]. By enforcing the principle of *separation of concerns*, this uniform interface creates a firewall between a mechanism's *clients* (*e.g.*, the application programmer or protocol implementer) and the mechanism's *provider* (*e.g.*, the mechanism implementer).

2. *Reuse via Inheritance* – as mechanisms are implemented in terms of C++ classes, the commonality of a set of mechanisms can be shared via inheritance. Using inheritance, a common base class provides the portion of the mechanism that is shared by all members of the set, and each mechanism is implemented by *deriving* a new sub-class from the base, which requires the implementer to provide only the portion of the mechanism that distinguishes the mechanism from the rest of the set. This technique allows both the reuse of interface (*e.g.*, several error reporting mechanisms which share the same interface for reporting which packets are missing) and the reuse of implementation (*e.g.*, several stream synchronization mechanisms which use the same underlying implementation for attaching new data streams).

3. *Rapid Prototyping* – utilizing the collection of protocol mechanisms provided with ADAPTIVE, combined with the techniques described above, protocol designers can rapidly develop new protocols by specifying the desired configuration of available mechanisms. Alternatively, protocol *mechanism* designers may use the library of available mechanisms as a reliable and consistent base with which new protocol mechanisms may be designed, prototyped and tested.

2.3 Adaptive Protocol Operation

In addition to the flexibility described above, a transport system must exhibit *adaptability* to sufficiently accommodate the dynamism that exist in both the application (*e.g.*, alternate coding schemes based on subject activity, adding/subtracting data streams or participants to a communication session) and the network (*e.g.*, bandwidth availability and packet loss rate fluctuations, latency variations due to a switch from terrestrial to satellite links). ADAPTIVE protocol configurations are capable of three classes of adaptivity:

1. *Parametric Adaptivity* – which varies the behavior of a communication session by adjusting some subset of its *parameters* (*e.g.*, inter-packet gap, transfer unit size, remote context update rate). Parametric adaptivity is suited to transient changes in the state of the network due to congestion as well as quantitative

changes in application behavior (*e.g.*, A/D sample rate increases/decreases).

2. *Functional Adaptivity* – which varies the behavior of a communication session by changing some subset of its *mechanisms* (*e.g.*, changing from selective repeat to go-back-n error recovery schemes, enabling/disabling gap or duplicate suppression). Functional adaptivity is required to adjust to fundamental changes in the state of the network due to sustained packet loss, increased latency, or changes in routing as well as qualitative changes in application behavior or requirements (*e.g.*, changing video coding schemes may require different error detection behavior, adding participants to a unicast data stream requires a different error recovery mechanism).

3. *Quantitative Adaptivity* – which varies the behavior of a communication session by adding or subtracting data streams. Quantitative adaptivity is required to accommodate multi-stream applications that selectively disable/enable multiple medium sources (*e.g.*, a teleconferencing application that switches from audio only to audio and video). Quantitative adaptivity is also required to accommodate multi-user collaborative applications that dynamically add or subtract participants from a workgroup.

ADAPTIVE provides various mechanisms which support the three types of transport system adaptivity described above:

1. *Explicit Mechanism Replacement Support* – by building protocol mechanisms based on uniform interfaces that hide the variance in different implementations, a single protocol mechanism (*e.g.*, update remote contexts) can be implemented by multiple different policies (*e.g.*, periodic updates, request-based updates). As described in Section 4, by providing explicit interface and implementation support for run-time mechanism replacement, ADAPTIVE offers an efficient and consistent framework for adaptive protocol operation.

2. *Application Feedback/Feedforward Control* – by providing callback mechanisms by which applications can be notified of changes in operating environment (*i.e.*, the network, the transport system, remote communication entities), and allowing manipulation of running session configurations via a uniform interface, protocol adaptivity can be placed under direct application control. As described in Section 3, ADAPTIVE provides this facility with multiple levels of granularity and scope.

3. *Network Feedback* – by utilizing information collected by the ADAPTIVE/UNITES subsystem, protocol adaptivity can be enabled by various conditions observed in the underlying network and local and remote transport systems. UNITES provides information on both the state of the network (*e.g.*, packet loss rate, channel utilization) and the state of local and remote ADAPTIVE entities (*e.g.*, buffer utilization, retransmission counts). Section 5 describe the metric collection facilities of ADAPTIVE in detail.

3 Map Applications and Networks To Transport Systems (MANTTS)

ADAPTIVE is a transformational system that configures and instantiates transport system configurations based on application requirements and network characteristics. The ADAPTIVE/MANTTS subsystem provides the Application Programmatic Interface (API) to the ADAPTIVE system through the use of ADAPTIVE Communication Descriptors (ACDs). ACDs provide a flexible mechanism for applications to describe (1) grade of service requirements, (2) application-transport system interactions, and (3) instrumentation/measurement configurations. MANTTS performs a series of transformations on an ACD to synthesize a Session Configuration Specification (SCS), which is used by the ADAPTIVE/TKO subsystem to instantiate and instrument a communication session.

3.1 Hierarchical Specification

Flexible and adaptive transport systems are of little utility if they lack an effective facility for applications to specify the characteristics required from a communication session. Various schemes for specifying an application's *quality* of service (QoS) requirements (*e.g.*, error rate, throughput, delay) as well as it's *functionality* of service (FoS) requirements (*e.g.*, connection-oriented vs. connectionless, best effort vs. acknowledged vs. reliable delivery) exist [21, 22, 23]. Existing schemes have been designed for transport systems that are either inflexible and/or non-adaptive to diversity in application or network characteristics. For a specification scheme to provide applications with an adequate interface to flexible and adaptive transport services, explicit support must be offered for the following:

1. *Variable Granularity* – both for QoS and FoS parameters, a specification scheme must provide fine grain control to applications that are aware of and require precise specification of a communication session configuration. Courser grain *macro*-level specification is also required for applications that are not aware of or are unconcerned with every detail of a session's configuration.

2. *Application-based Specification* – to allow most applications to specify a communication session in terms of the application domain, a specification scheme must provide sufficient insulation from the underlying protocol implementation. Application-based specification allows an application to specify high level communication *policies* (*e.g.*, deliver all data reliably) and relies on the transport system to decide on the actual *mechanisms* to be used (*e.g.*, PAR, ARQ, FEC).

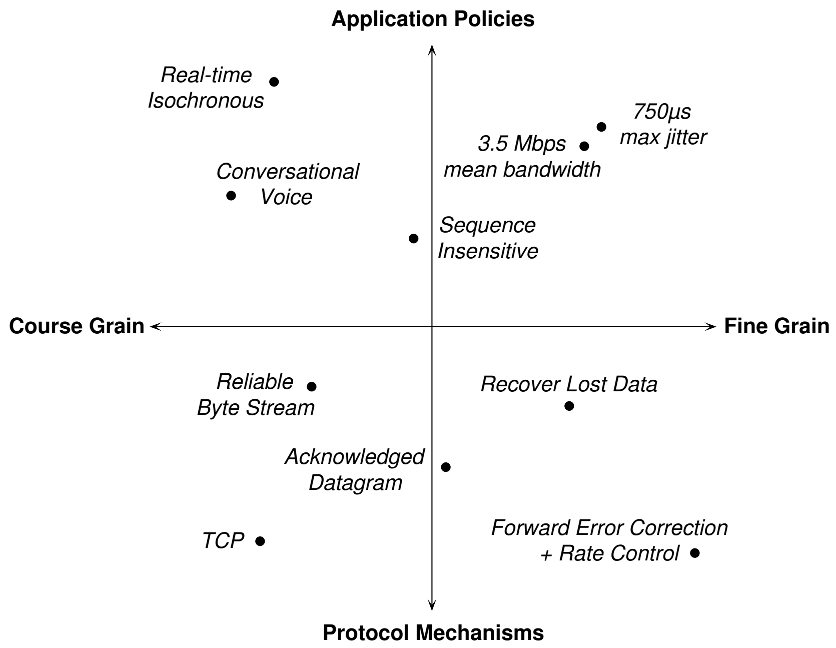

Figure 2: Hierarchical Specification

3. *Mechanism-based Specification* – provisions must be made for applications that require low-level control of the exact configuration of a communication session. Mechanism-based specification permits the application to bypass the application-based specification scheme by directly specifying the *mechanisms* used by a communication session. Provisions should be made for rejecting inconsistent protocol configurations resulting from incomplete or incompatible specifications.

4. *Application-Transport System Interaction* – the previous three requirements primarily address the initial configuration of a communication session. To effectively support the diversity and dynamism inherent in multimedia applications, explicit provisions for application-transport system interactions are required. These provisions take two forms:

   (a) *Transport System–Application Data Delivery*, which specifies the policies and mechanisms the transport system must use to deliver received data to the application. This entails dictating *when* to deliver the data (*e.g.*, immediately upon reception, periodically, or based on reception of related data) as well as *how* to deliver the data (*e.g.*, using an upcall mechanism[24] or read/write system calls).

   (b) *Application-guided Adaptation*, which specifies both the *conditions* the transport system needs to react to (*e.g.*, end-to-end delay exceeding some threshold, a remote application requesting an additional data stream on a connection) as well as the *actions* that are to be taken (*e.g.*, change retransmission mechanism, notify application via a callback).

The following section describes the ADAPTIVE Communication Descriptor, a hierarchical interface to flexible and adaptive transport services that satisfies the aforementioned requirements.

3.2 ADAPTIVE Communication Descriptor (ACD)

Applications request communication services from ADAPTIVE by providing MANTTS with a set of ADAPTIVE Communication Descriptors (ACDs). For a given communication session, the application furnishes a separate ACD per data stream that describes the behavior requested for that stream. Each ACD consists of five major components as follows:

1. **Quality of Service (QoS):** The QoS contains the *quantitative* description of the desired service. It allows the application to specify the a range of values to be used for each parameter (*e.g.*, minimum acceptable, expected maximum, expected mean, expected variance), providing a set of default values (*e.g.*, *don't care, maximum allowed, unknown*) for applications that are not capable of providing complete information. QoS parameters include throughput, connection duration, delay, jitter, and loss probability.

2. **Functionality of Service (FoS):** Applications specify the *qualitative* behavior of desired service using the FoS. The FoS describes the policies (*e.g.*, recover lost

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

Figure 3: Transport Service Classes

data, suppress duplicates, encryption) that are to be carried out by the transport system. In contrast to the QoS, which describes *when* and *how much* data will be transmitted, the FoS describes *what* processing must be done before transmission and reception.

3. **Data Synchronization and Delivery (DSD):** The DDS specifies the the policies to be used in synchronizing multiple data streams within one communication session (*i.e.*, what tolerance of intra-stream drift is acceptable, what action to take given a loss of synchronization), and the mechanisms to be used to ultimately deliver the data to the application (*e.g.*, via read/write calls, via upcalls into application, via in-kernel direct routing to a device).

4. **Transport Service Adjustment (TSA):** The TSA allows applications to participate in the dynamic configuration of the communication session. The TSA is a set of $< condition, action >$ pairs, where the *condition* specifies what event the application is interested in responding to, (*e.g.*, resource request denied, latency exceeds 10ms), and the *action* specifies the response to be taken, either as a callback to the application, or a call to an internal ADAPTIVE routine. These internal routines range from macro-level operations, (*e.g.*, abort connection) to very fine grain actions that implement both *parametric*, *functional*, and em quantitative adaptivity. MANTTS provides a special condition value that, when used in conjunction with these functional adaptivity operations, allows the application to "escape" the normal configuration process and hard-wire a protocol configuration. The session configuration that results from this direct specification method can then be validated by ADAPTIVE/MANTTS to guarantee that a meaningful protocol will be produced (*i.e.*, that fundamental mechanism incompatibilities do not exist).

5. **Transport Metric Configuration (TMC):** To accommodate protocol development, prototyping and measurement, the TMC allows the application to specify what performance metrics it is interested in monitoring. Each metric is specified by (1) *what* is to be measured and *where* (*e.g.*, host system throughput, per-stream transmission count, transmission delay), (2) the sampling and reporting rate (*e.g.*, sample every k milliseconds, report every n seconds), and (3) the reporting action to be taken (*e.g.*, add sample(s) to a repository, callback to application). The TMC allows any application using ADAPTIVE services to instrument a communication session.

3.3 MANTTS Operation

ADAPTIVE Communication Descriptors provide the API to ADAPTIVE. Once the ACDs have been created by the application and passed to ADAPTIVE, MANTTS must then transform these configuration requests that are expressed in terms of application-domain requirements into a *Session Configuration Specifier (SCS)* that can be used to directly instantiate a communication session. This transformation process examines the parameters of an ACD and attempts to match it to a pre-configured *Transport Service Class (TSC)* that represents a common set of communication requirements shared by a class of applications (*e.g.*, Real-Time Non-Isochronous, Interactive Isochronous). Figure 3 shows a representative set of transport service classes and the parameters they encompass.

4 Transport Kernel Objects (TKO)

Transport Kernel Objects (TKO) is a protocol composition framework that provides flexible data transport service to applications. It provides a set of uniform abstract interfaces and a library of mechanism implementations for the various functions required to compose multimedia communication protocols. TKO is implemented as a collection of C++ *classes* that allow protocols to be composed using *objects* that implement the mechanisms used in a particular protocol configuration. As shown in Figure 4, TKO consists of two major subsystems:

1. *TKO Operating Services Interface Library (TKO-OSIL)* – a set of C++ classes that provide an efficient uniform interface to the basic operating system services required by all protocols. TKO-OSIL allows protocols to be implemented in a portable and consistent manner.

2. *TKO Mechanism Class Library (TKO-MCL)* – a set of C++ classes that provide implementations of the various protocol mechanisms that comprise a communication session. TKO-MCL is partitioned into mechanism *families* that allow several alternate implementations of a protocol function to share both interface and implementation.

The remainder of this section describes these two subsystems.

4.1 TKO-Operating Services Interface Library (TKO-OSIL)

Implementing efficient communication protocols for general purpose computers requires operating system support for protocol development and operation. The services required from the operating environment include scheduling, buffer management, multiplexing/demultiplexing and context management. Most existing systems provide some subset of these services (*e.g.*, BSD-UNIX[3], UNIX System V STREAMS[2], *x*-kernel[1]), but with very little consistency across environments. For example, all three of the previously mentioned systems provide some form of buffer management (*e.g.*, BSD `mbuf`, STREAMS `mblk`, *x*-kernel `Msg`), but each has somewhat different semantics and interfaces for the basic set of operations (*e.g.*, logical vs physical copying, appending/truncating messages). TKO-OSIL provides a consistent interface to these basic operating system services for use by TKO-MCL protocol implementations by providing the following three C++ classes:

1. `TKEvent` – the basic abstraction for temporal events. Many protocols must respond to temporal events such as retransmission timer expiration or periodic update requests [25]. The `TKEvent` class defines an infrastructure for event management, providing operations like `TKEvent::schedule`, `TKEvent::happen`, and `TKEvent::cancel`. `TKEvent` objects schedule themselves to happen one or more times (*i.e.*, they are intermittent or periodic), they may be cancelled, and they are triggered to happen asynchronously by the operating system's timer facility. To accommodate the synchronization of multimedia applications and protocols to isochronous devices (*e.g.*, D/A converters, frame buffers), `TKEvent` allows periodic events to enable or disable drift compensation to overcome fluctuations in system scheduling services.

2. `TKMessage` – the basic abstraction for incoming and outgoing network messages. Previous work has shown that memory-to-memory copying is a significant source of transport system overhead [26]. Therefore, some form of buffer management is necessary to avoid unnecessary copying when moving messages between protocol entities and when adding or deleting headers and trailers [27]. The `TKMessage` class provides a uniform interface for services that create, copy, prepend, and split messages. `TKMessage` objects are internally divided into two distinct regions: the *header* and the *data*. The data region supports efficient logical copying operations and segmenting and reassembling of data chunks. The header region supports operations (*e.g.*, `TKMessage::prepend` and `TKMessage::unprepend`) that efficiently prepend header information onto a message and later strip it off. Explicit support is also provided for combining/separating component sub-messages belonging to multiple data streams for subsequent delivery to the network or application.

3. `TKSession` – the basic abstraction for a communication session. A protocol implementation must retain a collection of state variables for the proper operation of the protocol. In a multiprotocol environment, this includes both (1) information that a protocol must maintain on a per-session basis for addressing, internal buffer management, and protocol specific operations, and (2) some mechanism for associating the state variables of a session to the global state of the specific protocol the session is associated with, specifically, which *operations* or *methods* are to be performed as part of the protocol processing. `TKSession` encapsulates this information behind a uniform abstract interface that allows basic protocol operations (*e.g.*, `TKSession::send`, `TKSession::recv`, `TKSession::control`) to be properly dispatched to the appropriate protocol function. `TKSession` is implemented in two parts: (1) *global variables and functions* – responsible for `TKSession` creation and management and demultiplexing incoming `TKMessages` to the appropriate `TKSession`, and (2) *instance variable and functions* – responsible for performing the protocol specific operations on incoming and outgoing `TKMessages`.

These three classes provide the foundation for the operation and composition of protocols using the TKO Mechanism Class Library described below.

4.2 TKO Mechanism Class Library (TKO-MCL)

TKO-MCL is implemented as a C++ class library of reusable C++ protocol mechanisms. Each TKO-MCL class is an implementation of a single protocol function (*e.g.*, error detection, encryption, transmission control), that encapsulates both the *state* and *method* needed to perform the desired function. A TKO protocol is composed from multiple lightweight TKO-MCL objects, each of which performs a different protocol function. The remainder of this section provides a description of TKO-MCL and discusses several performance enhancements available to TKO-MCL protocol implementors.

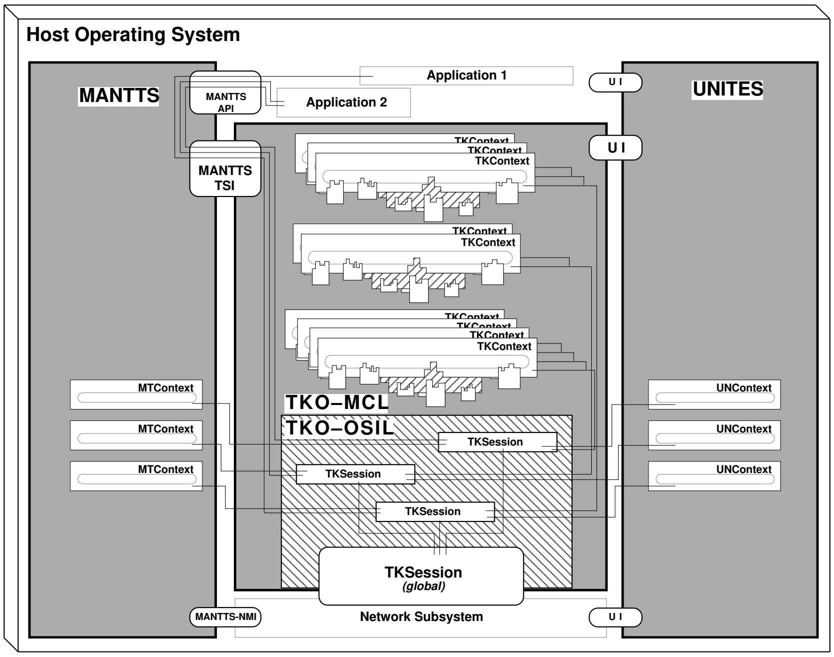

Figure 4: Transport Kernel Objects

4.2.1 Mechanism Families

To efficiently support flexible configuration and adaptive reconfiguration, TKO-MCL is organized as a C++ inheritance hierarchy. TKO-MCL takes advantage of C++ language mechanisms for (1) *encapsulation* to bind operations and their associated context allowing object-oriented protocol composition, (2) *dynamic binding* to allow protocol operations to be transparently and automatically selected at run-time, and (3) *inheritance* which allows multiple protocol mechanisms to be implemented as specializations of a single mechanism. As shown in Figure 5, the TKO-MCL class hierarchy is partitioned into multiple *Mechanism Families*, each of which provides one or more implementations of a given protocol function (*e.g.*, error reporting, encoding, stream synchronization). Each Mechanism Family consists of two distinct types of classes, a single Abstract Base Class (ABC), that defines the interface or *signature* for the protocol mechanism and optionally implements any shared or default behavior, from which one or more Concrete Derived Classes (CDCs) are derived, each of which represents a particular implementation of the abstract protocol function its family represents. Within a Mechanism Family, new mechanism implementations are usually implemented by deriving directly from the Abstract Base Class, but can alternatively be derived indirectly via a Concrete Derived Class when only a small amount of behavior in an existing implementation needs to be changed. Using derivation or *subclassing* as an implementation technique offers the following advantages:

- Shared interfaces allow multiple implementations to be transparently "plugged in" to perform a given protocol function. This interface consists of a collection of methods or C++ *member functions* that provide consistent and controlled access to the services provided by a mechanism implementation. The dynamic binding of *virtual* member functions in C++ ensures that the appropriate code is executed based upon the class a particular implementation is an instance of. A more detailed discussion of this appears in Section 4.2.2.

- Shared implementations allow a mechanism implementation to be expressed in terms of its *differences* from its base class. Reusing existing implementations via specialization allows new protocols to be implemented more rapidly and aids the task of protocol maintenance, as defects that are repaired in a base class are automatically repaired in any derived classes.

As shown in Figure 5, TKO-MCL provides a standard set of Mechanism Families that correspond to the basic mechanisms used in protocol processing (*e.g.*, Connection Management, Remote Context Management, Reliability Management, Stream Synchronization Management, Transmission Management). The Reliability Management Mechanism Family shown in Figure 5 is an example of a *Composite Component*, which is described in Section 4.2.3. As described above, each family contains a single Abstract Base Class that defines the basic interface to the mechanism, and multiple Concrete Derived Classes, that represent specific policy decisions that are used to implement a given mechanism.

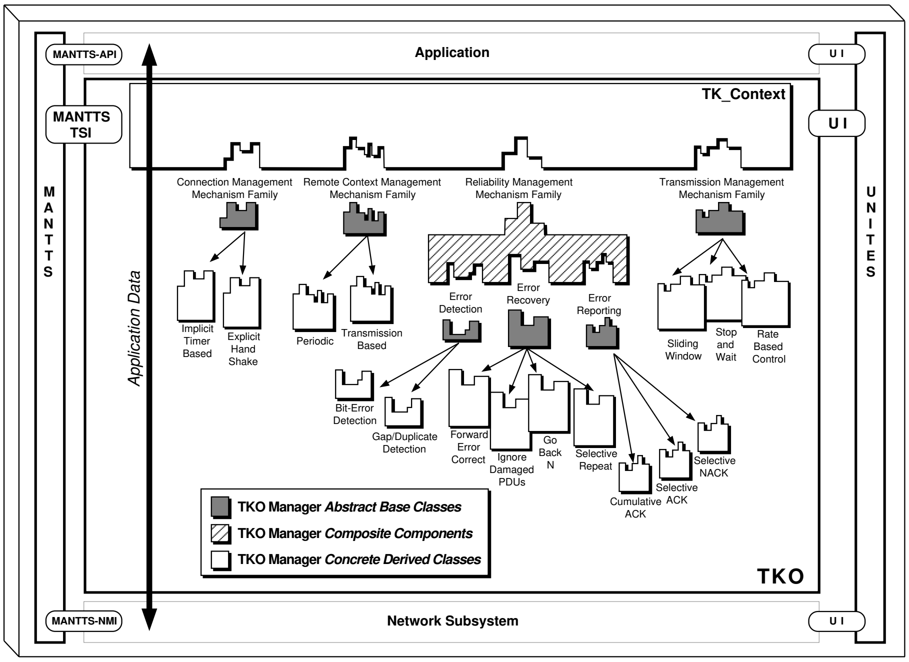

Figure 5: TKO Mechanism Class Library

4.2.2 Context Architecture

TKO provides an additional C++ class, `TKContext`, that links together a selection of various TKO-MCL mechanism implementations to form a cohesive protocol. TKO utilizes one `TKContext` per unidirectional data stream, combining multiple, possibly different, `TKContexts` to form a communication session. As shown in Figure 4, a single `TKSession` object is used to provide a rendezvous point for managing the `TKContexts` associated with the multiple data streams attached to a session. Each `TKContext` maintains a set of pointers to Abstract Base Classes, one for each TKO-MCL Mechanism Family. Operational `TKContexts` are created by setting these pointers to instances the appropriate Concrete Derived Classes.

4.2.3 Optimizations

As previously described, `TKContexts` maintain pointers to base classes and rely on language mechanisms to dynamically bind the appropriate executable code at run-time. Although studies have shown that it is possible to efficiently implement operating systems and communication protocols using these techniques [28, 29], ADAPTIVE/TKO provides several optimizations that streamline the creation and operation of commonly instantiated protocol configurations.

Composites: Figure 5 shows that Reliability Management is implemented as a *Composite Component*. Composites allow multiple related mechanism families to be bundled together into one larger mechanism. Composites are useful for enforcing relationships between multiple sub-mechanisms (*e.g.*, requiring go-back-N error recovery to use cumulative acknowledgments) while still allowing the sub-mechanisms to be independently used elsewhere (*e.g.*, using cumulative acknowledgment with sliding window flow control). Composites also allow the larger mechanism they represent to be replaced in one operation, which greatly reduces the complexity of run-time reconfiguration.

Preconfigured Contexts: TKO allows entire `TKContexts` to be preconfigured for commonly used protocol configurations. This entails implementing the class as a collection of actual instances of the constituent mechanisms, instead of a collection of *pointers* to instances that must be created separately and linked to the `TKContext` at run-time. This preconfiguration technique offers encreased performance by (1) eliminating one to two levels of indirection due to the pointer dereference and virtual function resolution and (2) by allowing instances of preconfigured contexts to be cached for faster instantiation.

5 UNIform Transport and Evaluation Subsystem (UNITES)

One of the primary goals of the ADAPTIVE system is to provide a framework for controlled protocol experimentation. ADAPTIVE provides an integrated experimentation environment by utilizing UNITES' metric specification, collection, analysis and presentation facilities. Performance data gathered by UNITES can be used to evaluate various protocol mechanisms and configurations with respect to (1) the level of service provided to the application, (2) the utilization of the underlying communication channel, and (3) the internal

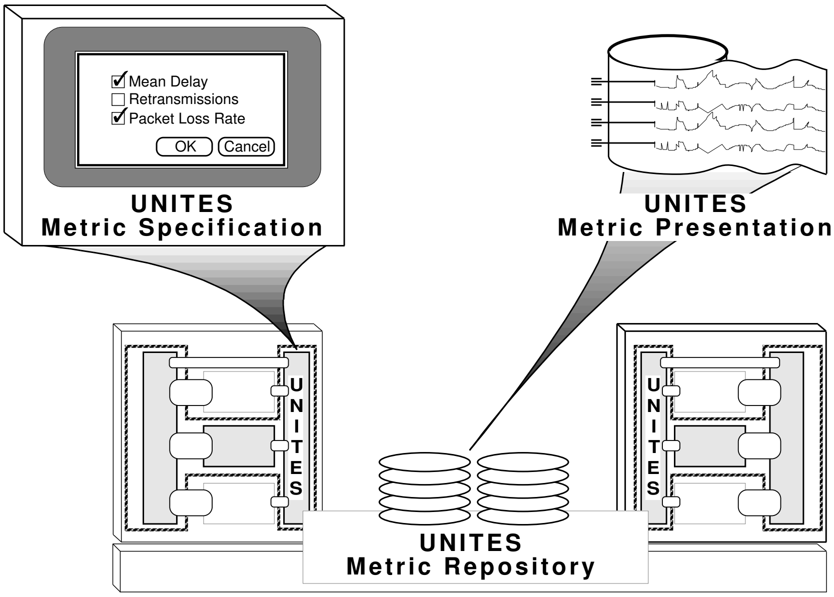

Figure 6: UNITES Architecture

performance characteristics of a given set of protocol mechanisms.

As shown in Figure 6, the UNITES Metric Repository stores the collected performance data in a shared database to minimize the intrusion made by the metric collection process [30]. Users may access this information via (1) UNITES-provided interactive graphic displays, (2) the UNITES C++ run-time library, or (3) standard network management protocols such as SNMP or CMIP. This metric data is available on either a systemwide, per-host, or per-connection basis. The performance monitoring process can be initiated when application programs use the Transport Measurement Component (TMC) parameter in the ADAPTIVE Communication Descriptor (ACD) to indicate the metrics they are interested in monitoring. ADAPTIVE then selectively instruments the instantiated TKO configurations and automatically collects the performance data during the operation of the system.

Metric collection may also be specified independent of a communication session using either a graphics-based or language-based interface to UNITES. Sjodin *et al.* [31] defines a specification language that indicates what measurements to collect and what traffic to generate. UNITES provides similar functionality with its UNITES Metric Specification Language (UMSL), but also provides a graphical interface that allows complex metric collection configurations to be specified using common user-interface elements (*e.g.*, check boxes, edit text fields, buttons, menus). This interface can be used to generate UMSL code for subsequent modification or to be used directly to configure a UNITES metric collection configuration.

UNITES supports two primary classes of metrics, *black box* and *white box*. Black box metrics require no knowledge of or interaction with the internal implementation of a protocol configuration. Black box metrics include application-based and host system-based metrics (*e.g.*, throughput, latency, and jitter) and network based metrics (*e.g.*, bit error rates, network utilization, and packet lengths). White box metrics require internal instrumentation of a protocol configuration and may be collected on a mechanism, mechanism family, connection, application, host system or system-wide basis. White box metrics include retransmission count, buffer utilization, instruction length, and scheduling and dispatching overhead. Both black box and white box metrics contribute to pinpointing performance bottlenecks in protocol configurations.

6 Summary

ADAPTIVE provides an integrated framework for protocol composition, evaluation and experimentation. It utilizes object-oriented design and implementation techniques to create an infrastructure for protocol composition that allows both flexible configuration and adaptive reconfiguration of communication protocols. The ADAPTIVE system integrates the hierarchical specification of application requirements and protocol configurations with the monitoring and reporting of performance metrics to create a transport system capable of adapting to network and application diversity and dynamism.

We are currently designing and implementing a prototype implementation written in C++ that runs under System V STREAMS. We plan to use this 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 [32], DQDB, and FDDI).

Acknowledgments

We would like to thank Hung Huang, Unmesh Rathi, and Girish Kotmire for their help in designing the ADAPTIVE system.

References

[1] N. C. Hutchinson and L. L. Peterson, "Design of the *x*-Kernel," in *Proceedings of the SIGCOMM '88 Symposium*, (Stanford, Calif.), pp. 65–75, August 1988.

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

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

[4] 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.

[5] M. Zitterbart, "Parallel Protocol Implementations on Transputers – Experiences with OSI TP4, OSI CLNP, and XTP," in *Proceedings of the First IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Feb. 1992.

[6] 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.

[7] 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.

[8] P. Steenkiste, "Analysis of the Nectar Communication Processor," in *Proceedings of the First IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Feb. 1992.

[9] M. L. Bailey, M. A. Pagels, and L. L. Peterson, "The *x*-chip: An Experiment in Hardware Demultiplexing," in *Proceedings of the First IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Feb. 1992.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] D. C. Feldmeier and E. W. Biersack, "Comparison of Error Control Protocols for High Bandwidth-Delay Product Networks," in *Protocols for High-Speed Networks II*, IFIP, 1991. a.

[16] B. W. Meister, "A Performance Study of the ISO Transport Protocol," *IEEE Transactions on Computers*, vol. 40, pp. 253–262, March 1991.

[17] L. Svobodova, "Measured Performance of Transport Service in LANs," in *Computer Networks and ISDN Systems*, 1989.

[18] B. Meyer, *Object Oriented Software Construction*. Englewood Cliffs, NJ: Prentice Hall, 1989.

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

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

[21] M. Zitterbart and B. Stiller, "A Concept for A Flexible High Performance Transport System," in *Telecommunications and Multimedia Applications in Computer Science*, pp. 365–374, Oct. 1991.

[22] CCITT, Geneva, *CCITT Recommendation I.211: B-ISDN Service Aspects*, 1991.

[23] EC, "IBC Application Analysis, research and Developement in Advanced Communications Technologies in Europe," Tech. Rep. R1071, Commission of the European Communities, xxx 1991.

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

[26] 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.

[27] S. W. O'Malley, M. B. Abbott, N. C. Hutchinson, and L. L. Peterson, "A Transparent Blast Facility," *Journal of Internetworking*, vol. 1, December 1990.

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

[29] 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.

[30] O. Endriss, M. Steinbrunn, and M. Zitterbart, "NETMON-II a monitoring tool for distributed and multiprocessor systems," *Performance Evaluation*, pp. 191–201, 1991.

[31] 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.

[32] 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.