A Programmable Transport Architecture with QoS Guarantees

Jean-François Huard and Aurel A. Lazar
Xbind, Inc.

ABSTRACT The emergence of distributed multimedia applications exhibiting significantly more stringent quality of service requirements than conventional data-oriented applications calls for new transport protocols with different characteristics to coexist and be integrated within single applications. The different delivery requirements posed by these diverse multimedia applications often imply the need for highly customized protocol implementations. Hence, application developers are faced with the threat of code obsolescence caused by the development of even newer delivery techniques. We present an object-oriented transport architecture that allows for dynamically binding a variety of protocol stacks on a per-call basis. By binding protocol stacks together, the special needs of the application can be met without the need to rewrite the code. This differs significantly from the traditional transport architecture which assumes preinstalled transport protocol stacks that cannot be customized. To illustrate some of the advantages provided by the architecture, we describe the transport component of the first reference implementation of the ISO MPEG-4 Delivery Multimedia Integration Framework and demonstrate how quickly it was implemented in our framework.

The emergence of distributed multimedia applications exhibiting significantly more stringent quality of service (QoS) requirements than conventional data-oriented applications calls for new transport protocols with different characteristics to coexist and be integrated within single applications. The different delivery requirements posed by these diverse multimedia applications often imply the need for highly customized protocol implementations. Hence, application developers are faced with the threat of code obsolescence caused by the development of even newer delivery techniques. Furthermore, application developers have to contend with the fact that a single protocol stack may be insufficient to meet all their needs.

In order to address this challenge, we propose an object-oriented transport architecture in which the atomic processing entity is based on the consumer/producer paradigm. The architecture consists of consumer/producer components that are separately represented by their transport abstraction, called an *engine*, their control and management abstractions, called front-ends, and a set of controllers implementing network services such as the dynamic binding of suitable protocol stacks.

The consumer/producer engines are software components located in the data path set up by the multimedia communication session and are responsible for data processing (including protocol implementation), multiplexing, and scheduling of channels. The engines also interact with the operating system when transferring media flows through network interface cards.

The control and management front-ends define a model for controlling and managing the consumer/producer engine resources (e.g., resource provisioning, accounting, and QoS control). The latter includes QoS monitoring, QoS violation detection, QoS adaptation, and QoS renegotiation.

Controllers provide network services to the application programmer and thereby offload work from applications. By encapsulating domain-specific knowledge, they reduce the amount of technical knowledge required by the application developer for creating multimedia applications.

The protocol stack is modeled in an object-oriented manner. The protocol stack builder is responsible for the construction of the protocol stack by binding a variety of engines together. It is aware of the variety of transport components supported on the end system and the order in which they can be bound together to create useful protocol stacks. The protocol stack builder performs the dynamic binding of the components at runtime and creates a suitable transport protocol stack according to application QoS requirements. It allows, on a per-call basis, dynamic binding of a variety of protocol stacks that are tailored to the special needs of the application. The control and management functionality of each producer/consumer is abstracted via a standardized interface. This allows the protocol stack builder to select and bind a set of consumer/producers and to create meaningful protocol stacks at runtime. The consumer/producer engines run in user space. This architecture differs significantly from the traditional transport architecture, which assumes preinstalled transport protocol stacks that cannot be customized.

To illustrate some of the advantages provided by the architecture, we describe the transport component of the first reference implementation of the ISO MPEG-4 Delivery Multimedia Integration Framework (DMIF) and demonstrate how quickly it was implemented in our framework.

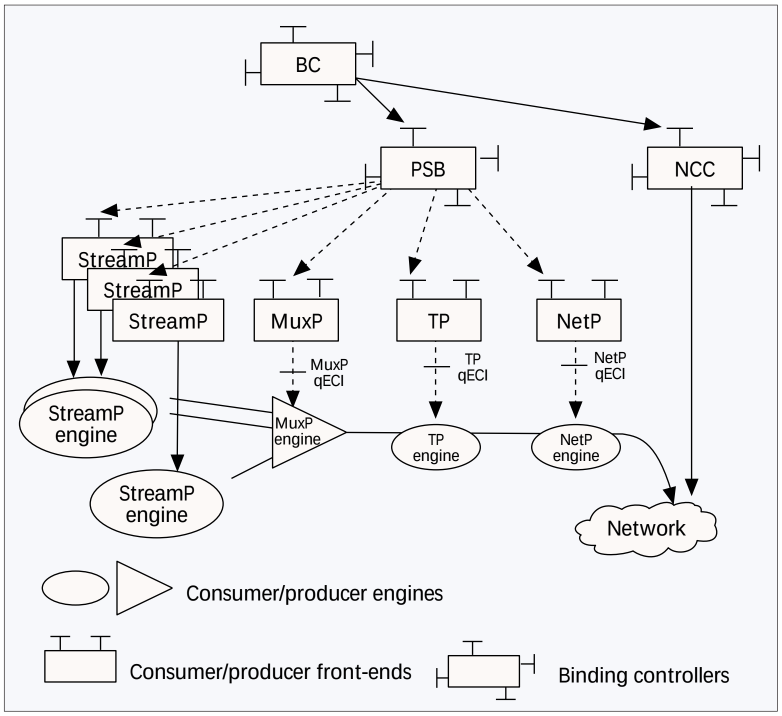
This article is organized as follows. In the next section the programmable transport architecture is described. The media transporter components are discussed after that. We present some implementation considerations concerning the interaction mechanisms between the various architecture components and describe the transport component of XDMIF. Finally, related work is reviewed.

ARCHITECTURE

The key atomic entity of the architecture is based on the consumer/producer model. The consumer/producer components can be classified into two categories: *media processors* and *media transporters*.

Media processors refer to components that transform media streams from one format to another by processing the content of the stream. Examples of these include transcoders and encryption devices. The current architecture defines two media processors, namely, the streamed device that abstracts media stream producers and consumers, and the encryptor. The media stream producers grab data via a physical device (e.g., camera or microphone) and compress it into encoded data (e.g., a video board that compresses raw video into MPEG-2 encoded video). The media stream consumers render the data to a physical device (e.g., display or speaker) after having decoded the compressed data; for example, a digital signal processing (DSP) chip set that processes encoded audio and plays it back. Finally, the encryptor either encrypts or decrypts data depending on whether it is located on the sender or receiver side of the communication channel.

Media transporters carry and route media streams without discerning or altering their contents. They implement the transport functionality such as scheduling of channels, multiplexing, flow control, encapsulation, and deencapsulation, segmentation and reassembly, and so on. On the transmitting side of a channel, media transporters add a header to each data unit sent to the receiving peer. The transmitting side may also fragment the message if it is too large to be transferred to the following media transporter in the protocol stack. On the receiving side, media transporters reassemble messages, and deencapsulate and forward data units to the next component in the protocol stack. A chain of media transporters is traversed until the data reaches a media processor such as a streamed device that renders it back or saves it in a file for future use.



■ **Figure 1.** *Protocol stack using the consumer/producer components of the architecture.*

In this article, the emphasis is on the media transporter components of the architecture. Although the design of the media processor components is identical to the design of the media transporters, the functionality of the media processors is not addressed. The media transporter components are described in detail later. First, however, the generic description of the consumer/producer model consisting of the consumer/producer engine and consumer/producer front-ends as well as the controllers acting on the front-ends is presented.

OVERVIEW

The consumer/producers shown in Fig. 1 are separately represented by their transport abstraction, called an *engine*, and their control and management abstractions, called *front-ends*. Examples are the NetP engine and NetP, the TP engine and TP, and the MuxP engine and MuxP, respectively. By directly acting on the latter abstractions, a binding controller (the protocol stack builder, PSB, in Fig. 1) manages and controls the generation, consumption, and processing of streams without being located in the data path.

The consumer/producer engines (CPGs) are responsible for data processing (including protocol implementation), multiplexing, buffering, and scheduling of channels. CPGs are located in the data flow path of a multimedia communication session and are the actual components that process and transport the data. Their implementation is generally platform-dependent. Two examples of engines are *qStack* [1], a lightweight transport protocol for real-time interactive communications, and an MPEG-2 video encoder that compresses a raw video bitstream into a lower-bit-rate encoded video stream.

The consumer/producer front-ends (CPFs) define an abstraction layer for the signaling system to remotely control and manage CPGs. For example, in Fig. 1 TP abstracts the control and management interfaces of the transport protocol component of a protocol stack. TP controls the TP engine via a transport protocol QoS-based engine control interface (TP qECI). A variety of TP engines can be implemented (e.g., *qStack*, TCP, RTP) as long as they provide the TP qECI interface to the TP.

Binding controllers (BCs) allow the creation, operation, management, and programming of services. Their primary purpose is to offload work from the application. BCs elevate the level of abstraction needed to develop multimedia applications by encapsulating domain-specific knowledge, thus reducing the amount of technical knowledge required by the application developers to write multimedia applications. Examples of such controllers are the PSB and network connection controller (NCC), as shown in Fig. 1.

Figure 1 illustrates some of the interactions and components involved during the creation of a multimedia service. Only a subset of the interactions, objects, and interfaces are shown. Three layers of components are represented. The engines are located in the data path at the bottom layer. The front-ends allow for media control and management abstractions and are located in the middle layer. The binding controllers are located on top. The PSB creates, binds, and controls media processors and media transporters. Finally, the NCC creates network connections between end systems.

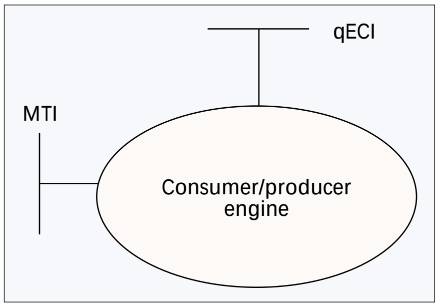
The architecture allows for the dynamic creation of protocol stacks by binding engines at runtime. The architecture further allows complex multimedia stream processing operations to be constructed by cascading media processor engines. Finally, the architecture allows for multiple implementations of a media transporter engine to coexist and for new capabilities (e.g., new consumer/producers) to be added without having to modify any of the components of the architecture. This differs significantly from the traditional transport architecture which assumes preinstalled transport protocol stacks that cannot be customized.

CONSUMER/PRODUCER ENGINES

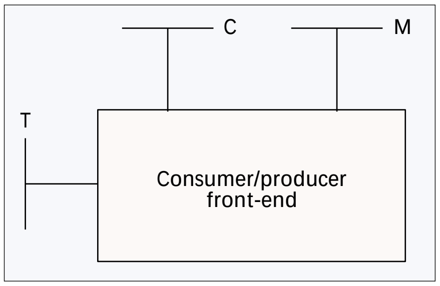
Engines are software components that implement protocol state machines with the capability of supporting and scheduling multiple channels at once. They are implemented in user space and are generally hardware-independent but strongly dependent on operating system multithread programming support. The CPGs are located in the communication data path; their representation is illustrated in Fig. 2.

An engine has two interfaces: a qECI and a media transfer interface (MTI). The interface represented on top of the engine is the qECI. It is visible only to the front-ends that are bound to it and using its specific services. Engine interfaces are named using the front-end acronym followed by qECI; e.g., the transport protocol engine control interface is denoted TP qECI.

The interface represented on the side of the engine is its MTI. It is used to move the data from one CPG to another. Additionally, the MTI provides some I/O control capability, to query an engine's maximum service and protocol data unit sizes, set the blocking mode, flush the engine's buffer, and so on. The engine's MTI is visible only to the adjacent engines in the data path. Only one interface represented as the same is used to both send and receive data from an adjacent engine. An opaque handle (a number identifying the channel) is provided to distinguish between each channel the engine processes. Finally, as opposed to the qECI which is specialized for each engine, the MTI is common to all engines; therefore, any engine can be connected to any other engine, and any protocol stack can be built as desired.



■ **Figure 2.** *A consumer/producer engine with its qECI and MTI interfaces.*



■ **Figure 3.** *Consumer/producer front-end with its control (C), management (M), and transport (T) interfaces.*

When a data unit is transferred using an MTI method (e.g., via a `send()` or `recv()`), the engine processes the data unit and schedules its transfer to the adjacent engine in the data path. In the case of the network provisioning engine, it schedules the data unit to be sent out into the network. Appropriate scheduling and the elimination of unnecessary data copies have to be carefully considered in order to achieve satisfactory overall performance and throughput.

CONSUMER/PRODUCER FRONT-ENDS

The CPF defines an abstraction layer for the signaling system to control the heterogeneous transport and computing resources of the end system. Front-ends have open interfaces, and each front-end abstracts a single segment of an end-to-end channel. An end-to-end channel consists, therefore, of multiple segments, one for each engine it traverses. Abstracting each channel segment allows for each of the channel's protocol stack state machines to be controlled. Multiple front-ends can be attached to a single engine since each engine has the capability to support multiple channels and their scheduling.

The representation of a CPF is shown in Fig. 3. The control and management interfaces (labeled C and M, respectively) are drawn on top of the object. As their name suggests, they provide capabilities for controlling and managing the channels they abstract, such as resource provisioning, accounting, and QoS control (e.g., QoS monitoring, QoS violation detection, QoS adaptation, and QoS renegotiation). The front-ends management capabilities are to dynamically load engines, to obtain an engine's MTI, and to obtain the handle of the associated channel segment. The transport interface (labeled T) is used for binding and the negotiation of media transfer parameters (e.g., maximum protocol data unit size).

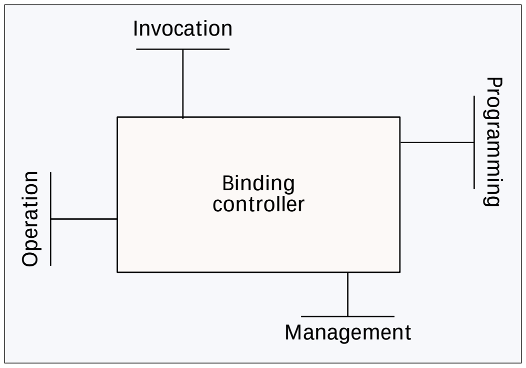
The front-end is a signaling abstraction which provides a way of connecting a remote client (e.g., a binding controller) with the low-level service that implements the transport functionality of the consumer/producer. In order to provide transparent access for a front- end's client to the engines, each engine must support the functions defined by its qECI. The front-end's client and engines do not need to use the same language in order to communicate since they have access to their respective middleware front-end interface. Operations invoked at the front-end are interpreted and forwarded to the engine using the engine's qECI. Most of the control interface (C interface) operations are mapped to a qECI operation, and the M interface is mainly used to manage the front-end. Examples of C interface operations are the establishment and release of a channel. As for the M interface, methods for loading an engine and configuring it are available.

The interfaces of the collection of CPFs exposing states to binding controllers form a repository called the binding interface base (BIB) [2]. The BIB consists of QoS-based application programming interfaces (APIs) used by the binding controllers and management system to control and manage the engines. The front-end interfaces are open, thus allowing for end-system transport and computing resources to be remotely controlled.

The front-ends use dynamically linked libraries that allow them to be bound to engines at runtime. In this way, a protocol stack can be composed and built dynamically. A front-end needs to only know the definition of the type of engine it abstracts in order to be bound to it.

BINDING CONTROLLERS

As opposed to the front-ends that represent the low-level middleware services, the BCs represent high-level middleware controllers that an application developer would normally use to build a multimedia application. Examples of such controllers are network connection control, transport monitoring, QoS parameters translation, and protocol stack building. Collectively, the BCs provide broadband kernel services [2]. A representative BC is illustrated in Fig. 4.



■ **Figure 4.** *A binding controller with its four interfaces.*

A BC is composed of an algorithmic component and a data component [1]. The algorithmic component expresses the execution logic of the service instance, while the data portion is an abstraction of its state. Controllers have four interfaces that allow the creation, operation, management. and programming of the controller. The interfaces are open and allow all binding controllers to be remotely invoked.

The service invocation interface is the entry point of the execution or instantiation of a service. The service operation interface defines the operational functionality of the controller and allows for monitoring and manipulation of service instance states during execution. It is typically the primary interface of the controller. The interface for programming services allows manipulation of the logic to be performed while a service is in execution. Finally, the service management interface allows for monitoring of the controller states and manipulation of controller parameters.

To illustrate the use of each interface, let us consider the service provided by the PSB. The service invocation interface is the entry point for creating a useful protocol stack based on the QoS requirements and a given network endpoint, such as a virtual path identifier/virtual connection identifier (VPI/VCI) pair in an asynchronous transfer mode (ATM) connection. The operation interface allows changing the QoS associated with a protocol stack or even modifying the protocol stack while it is active. The programming interface allows the management system to configure the logic of the PSB, that is, the rules on how to create useful protocol stacks. Finally, the service management interface allows the management system to manage the PSB, for example, by specifying the maximum number of transport channels it may allow.

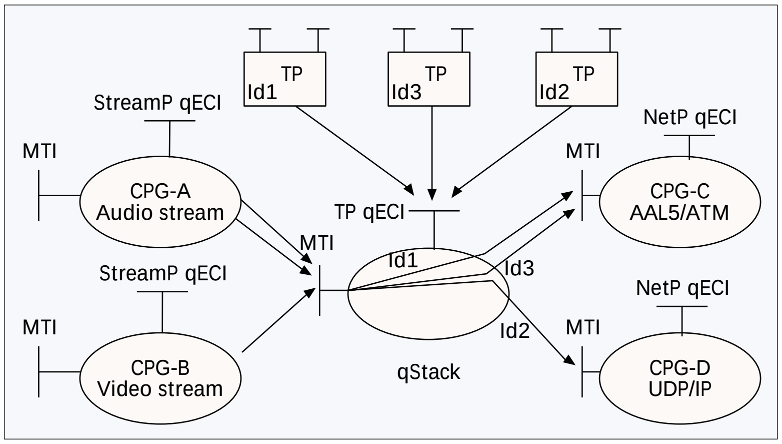


Figure 5. *Interactions between consumer/producer front-ends and engines.*

INTERACTIONS BETWEEN CONSUMER/PRODUCER ENGINES AND FRONT-ENDS

The purpose of distinguishing between engines and front-ends is for separating the control on two different timescales: that of transport (i.e., at the packet level) and that of flows or virtual circuits (i.e., at the call level). Furthermore, the separation allows for remote control of the individual transport channels. As mentioned above, each engine has capabilities of software multiplexing, and thus, may support multiple channels, each abstracted by a single front-end. The latter contains the specific state of the channel (e.g., the required and measured QoS). When channel segments are initialized, the engine assigns an opaque handle to access the channel's state and provides the handle to the front-end. All front-ends that are bound to an engine can access the capabilities of the engine via the engine's qECI and the opaque channel's handle.

Figure 5 illustrates the interaction model between CPGs and CPFs. The engine under consideration is in the middle and is bound to four other engines (CPGs A through D). The middle engine supports three channels, each abstracted by a CPF. In order to better illustrate the example, let's assume that CPG A and CPG B are audio and video streamed devices, CPG C and CPG D are AAL5/ATM and UDP/IP network provisioning engines, and the middle engine represents the *qStack* transport protocol engine. *qStack* provides flow control and QoS support for two audio channel segments, one going through an IP network and one through an ATM network, and for a video channel segment that goes through an ATM network.

The three channel segments of the middle engine are labeled Id1, Id2, and Id3. Each of the three front-ends bound to it abstracts and controls one channel segment and locally maintains the opaque handle that identifies these. The audio device connected to the ATM network has its *qStack* segment labeled Id1. The data path is composed of CPG A, middle engine, and CPG C. The *qStack* channel segment is controlled by the left CPF with identifier Id1. CPG A also maintains the identifier Id1 needed to provide data to *qStack* through the MTI. The identifier was provided when the data path was created by the PSB. Segment Id2 is controlled by the right CPF. It abstracts the middle segment of the audio channelconnected to an IP network; that is, it consists of the data path composed of CPG A, middle engine, and CPG D. The last segment, with label Id3, is controlled by the middle CPF. Again, it is the middle segment of the data path, composed of three CPGs, namely, CPG B, middle engine, and CPG C. As can be seen, the middle engine performs protocol processing and is responsible for multiplexing channels between adjacent engines. The media transfer between engines is achieved via the MTI using the opaque handle that identifies the proper segment. For example, when CPG B invokes a media transfer operation at the *qStack* MTI, it provides the handle with Id3.

A detailed illustration of all the protocol stack components would have included six additional front-ends: two attached to CPG A for controlling the two audio streams, one attached to CPG B to control the video stream, two attached to CPG C to control the ATM connections, and one attached to CPG D to control the UDP/IP connection.

The next section will discuss some implementation issues related to the actual binding of the engines and front-ends.

MEDIA TRANSPORTER COMPONENTS

By specializing the interfaces of the CPGs and front-ends, the media processor and media transporter components of the protocol stack are defined. In this section three media transporter components and three transport services are described. The following subsection describes the functionality of the network provisioning media transporter. Similarly, the transport protocol and multiplexer media transporters are described in the two next subsections, respectively. The fourth subsection describes the transport services: the PSB, a QoS mapper that performs translation of QoS between different protocol stack layers; and a monitoring server that performs accounting for management purposes.

NETWORK PROVISIONING

The network provisioning (NetP) component is responsible for data transmission across networks. It provides transparent transfer of data for the transport protocol component (described in the next section). In the architecture, the network provisioning component provides the protocol stack base layer (i.e., the lowest level of the end system protocol stack). This layer includes the user plane functionality of traditional transport protocols such as UDP/IP and ATM adaptation layer type 5 (AAL5)/ATM.

The NetP control capability consists of the provisioning of calls (e.g., opening/closing virtual circuits) and network QoS control. The provisioning capability revolves around the creation and destruction of opaque network handles (e.g., sockets) associated with network endpoints. Its QoS control capability is related to pacing the injection of packets into the network and monitoring the packet QoS. The NetP management capability consists of setting network QoS requirements (when applicable) and accounting. The accounting capability consists of counting the number of transferred packets. The front-end does not directly perform the control capability but maps them to its engine, marshalling the arguments of calls whenever needed.

The control capability is typically realized by invoking system or library calls that communicate with a network interface card driver. For the pacing capability, if not provided by the hardware, interval timers might be used. However, pacing is expected to be provided by the network interface card since software pacing is inefficient.

The processing capability of the NetP engine is related to the per-byte (or per-bit) operations (e.g., checksum). The engines are expected to deliver error-free packets, but could potentially deliver packets containing bits in error, with a proper notification to its recipient. No assumption is made about the ordering of delivered packets. For the implementations of the ATM and IP NetP, the MTIs simply invoke the equivalent system or library calls after performing some additional protocol processing. Examples of protocols that the NetP engines provide are UDP/IP and AAL5/ATM.

Transport Protocol

The TP media transporter is used to provide end-to-end communication capability with QoS support. It is the first layer in the protocol stack that is end-to-end QoS-aware. If needed, the TP media transporter has the ability to ensure reliable end-to-end communication. It relies on network provisioning to receive error-free data, but may have to request retransmission if a segment of data is missing. Flow control resides in this layer to manage the data flow through the channel. This means the TP media transporter assumes that a feedback channel is available for QoS adaptation and flow control.

The TP defines the end-to-end QoS control, management, and accounting capability. The end-to-end QoS control API is for setting the parameters of the TP and for monitoring the delivered QoS. The specific capability of the control interface depends on the transport protocol specifications; however, the API is common for all TPs. The API can be used to set end-to-end QoS to be delivered to the application. The management interface allows for the selection of the TP engine associated with the channel. It can also be used to obtain accounting information such as the call duration and the amount of recovered data delivered to the application. The TP front-end also performs slow timescale monitoring of channels and initiates QoS renegotiation procedures upon detection of sustained QoS violations. Finally, the TP shall have the capability to dynamically switch engines (i.e., changing the TP algorithm) to adapt to large QoS variations. This capability is further discussed later.

The TP engine ensures the efficient delivery of data and performs the "in-flow" QoS monitoring, that is, monitoring on a fast timescale. It ensures that end-to-end QoS is provided to the next engine bound to it upstream. Every TP engine must implement flow control, encapsulation/decapsulation, scheduling of channels, segmentation and reassembly, buffering, multicasting, and QoS monitoring and QoS adaptation mechanisms.

Transport Multiplexer

The multiplexer (MuxP) media transporter provides the capability of multiplexing multiple streams into one channel. This is a useful component when multiple media processors of a single multimedia application are simultaneously active (e.g., DMIF FlexMux [3]). This capability is also useful when many short-term channels are opened and closed on a common host (e.g., Web browsing).

The MuxP engine performs the software multiplexing and demultiplexing of the channels that are established within a single transport channel. It is expected that all channels of the MuxP engine experience the same QoS. However, each channel may have different bandwidth requirements.

The MuxP front-end must ensure that the total capacity of the established channels is within the capacity region of the MuxP engine. Typically, MuxP is bound to a TP, but can also be directly bound to a NetP. MuxP provides the ability to add and destroy channels with a given QoS and ensures that the overall QoS budget of the channel is within the negotiated QoS of the TP to which it is bound. There is only one MuxP for all channels established through the multiplexer. This prevents the proliferation of front-ends and ensures scalability.

Transport Services

The previous sections introduced three kinds of consumer/producers located in the data path of the protocol stack. This section introduces three transport services that are provided in the architecture. The PSB creates protocol stacks by invoking the control and management interfaces of the NetP, TP, and MuxP. The transport monitor performs accounting for management purposes (i.e., monitoring of the amount and kind of data transferred and the channel holding time). Finally, the QoS mapper performs translation between QoS specifications of different protocol stack layers.

Protocol Stack Builder — As mentioned before, the protocol stack is modeled in an object-oriented manner. The PSB is responsible for constructing the transport sections of the data path by binding MuxP, TP, and NetP consumer/producers together. The PSB can be a centralized server, but is expected to be highly distributed (one on each end system). The PSB is aware of the variety of engines supported on the end systems and the order in which they have to be bound to create a useful protocol stack through its management interface. The PSB performs the dynamic binding of engines at runtime by attaching front-ends together. Its functionality is similar to that of a connection manager which creates a virtual channel in an ATM network, but does so on the end system.

Transport Monitor — The *transport monitor* is used to log accounting information for management purposes. Its interface is used by the front-ends to register connections and log QoS information. When the data path comprises a TP, only the TP registers. When no TP is used, the NetP registers its channel. When a front-end deregisters its channel, the accounting information is logged permanently for future usage by the management system (e.g., for billing, network performance, and dimensioning purposes).

Any type of information related to the amount of data transferred, duration of a call, and so on can be logged. Furthermore, the delivered and requested QoS can be logged. Finally, whenever QoS renegotiation is successfully performed with a TP, the new QoS is logged.

The transport monitor receives QoS monitoring information at regular intervals and evaluates QoS on a slow timescale compared to that on which the TP operates. When the transport monitor detects a large QoS variation for a channel, it may initiate a transport protocol renegotiation that can result in dynamically changing the TP engine at runtime.

QoS Mapper — The *QoS mapper* performs translation of QoS specifications between the various protocol stack levels (application, transport and network) [4]. Applications and PSBs at call establishment and QoS renegotiation time invoke its mapping capabilities. Unlike the PSB, which controls a variety of front-ends, the QoS mapper does not interact with them.

INTEGRATING THE ARCHITECTURE IN ITS RUNTIME ENVIRONMENT

As already mentioned, by distinguishing between the consumer/producer engines and front-ends, a natural separation between signaling (or control) and transport is achieved. Separation of control and transport is a well-established principle that can be recognized in most protocol stack designs and implementations. From the service creation and development standpoints, this separation allows for rapid development of multimedia applications because the signaling system and transport components can be independently realized. This turns out to be important during implementations since multiple groups can work on the development of the transport components in parallel without the need for constant coordination. In addition, code reuse is potentially greater since the various implementations of the media transporters are separated from the signaling implementations.

For example, during the *XDMIF* development (described later) the specification of the MuxP engine and front-end interfaces allowed rapid development of the software by two groups of developers working in parallel. Using a preliminary implementation of the engine, DMIF applications could be built while the transport components were developed. This should be contrasted with a monolithic approach to software development, which calls for application development to be postponed until all the transport components are made available.

PROTOCOL STRUCTURE

The structure of the architecture and its implementation differs significantly from traditional transport protocol systems. First, the runtime execution environment of the architecture is the user space. Second, each front-end is implemented as a class that inherits its protocol structure from a base protocol class called the `VirtualPort`. Each front-end is a specialization of the `VirtualPort`, and implements its own control functionality and reuses the binding functionality implemented in the base protocol class. The use of inheritance imposes the protocol structure and facilitates the dynamic construction of protocol stacks at runtime.

The engines are located in the data path, and therefore are optimized for speed, memory usage, and buffer management. They are implemented in the C programming language and built as dynamically loadable libraries. In contrast, front-ends are signaling entities that use Common Object Request Broker Architecture (CORBA) for communications with the binding controllers and are implemented in C++.

CPFs and CPGs run in the same address space. Engines are loaded by front-ends at runtime using the operating system dynamic library loading mechanisms. The engine interfaces (MTI and qECI) are implemented as procedure dispatch tables. Using dispatch tables serves two purposes. First, it is more advantageous for the front-ends since a single call can be made to discover the entire set of entry points. Second, this enables the layered services to be formed and operated more efficiently.

The functionality of the front-ends control API and engines qECI are equivalent. Every front-end control operation (e.g., `frontEnd->Op1()`) has its equivalent qECI interface (e.g., `engine->op1()`). It is the responsibility of the front-end to convert the CORBA parameters to platform-dependent parameters used by the qECI, and to invoke the qECI.

PROTOCOL STACK BUILDER

The PSB dynamically creates a variety of protocol stacks tailored to the special needs of the application on a per-call basis. It builds a specialized graph for each channel by instantiating the media transporter engines required for protocol processing in the data path. If the media transporter class is not yet loaded, it loads it into the service daemon. The daemon is able to support multiple engine implementations of each type of media transporter since each engine is loaded dynamically and provides the required common qECI. A binding controller (not discussed in this article) manages the context (session state) of each channel. The binding controller keeps the state of the communication session and can tear down the channel upon request to the PSB. The PSB is stateless and performs only protocol binding. For the QoS mapping, the QoS mapper must be invoked. The QoS mapper also resides in the service daemon.

PERFORMANCE CHALLENGES IMPOSED BY THE ARCHITECTURE

Implementing protocol stack components in user space introduces performance challenges that are easier to resolve with the traditional approach of implementing the transport functionality into the operating system. For instance, high-resolution timers are not always available in user space, jitter is introduced by context switching, buffer management might force a component to perform additional memory copies, or a channel scheduler must be implemented. The most difficult aspect of implementing protocols in user space is associated with the scheduling of channels. A responsive scheduler is needed on the receiver side to avoid introducing excessive latency. If the operating system message (or signal) mechanism is slow, excessive jitter can be introduced in the scheduling process.

However, having protocol module interfaces standardized and implemented in user space allows for the rapid development of new protocol engines that are better tailored to new media QoS requirements. Furthermore, user space implementations facilitate their deployment in the marketplace since users do not have to wait for operating system manufacturers to provide such protocols.

Performance measurements are not provided in this article and will be presented elsewhere. Suffice it to say, however, that a multimedia application may easily require up to 20 times more resources than the protocol implementation itself. Furthermore, simple measurements indicate that the full deployment of a protocol stack for an MPEG-2 application using a 6 Mb/s stream requires less than 3 percent of the CPU resources on a 233MHz Pentium PC running Windows NT.

User space implementations of lightweight TP might not be adequate for high-performance (specialized) servers, but are certainly adequate for multimedia end systems. The most commonly implemented user space protocol is probably RTP/RTCP. However, RTP is implemented as part of the application, and its flow control algorithm is tailored for the specific application needs. This considerably reduces the potential of code reuse.

ILLUSTRATION: XDMIF

In this section we illustrate some of the advantages provided by our transport architecture by describing the first reference implementation of the ISO MPEG-4 DMIF and demonstrating how quickly this standard and its QoS extension were implemented in our framework.

DMIF Overview

MPEG-4 is an emerging international standard that specifies the coding of audio and video data using object description techniques [3]. This novel approach to coding allows multimedia applications to dynamically compose complex scenes from one or more elementary multimedia streams or synthetic objects. The scene description information, consisting of the logical structure of the scene, spatial and temporal information of the video/audio objects that make up the scene, object attributes, and graphics primitives, is carried within potentially hundreds of data channels. This calls for the establishment and release of numerous short-term channels with the appropriate QoS at a high rate. Traditional methods of signaling are not adequate to meet this demand because of the high overhead introduced.

In addition to defining the format, structure, and rules of composition of these objects, MPEG-4 also specifies a general application and transport delivery framework called the Delivery Multimedia Integration Framework (DMIF). Specified by MPEG-4, DMIF's main purpose is to hide the details of the transport network from the user, as well as to ensure signaling and transport interoperability between end systems.

In order to keep the user unaware of the underlying transport details, DMIF defined an interface between the user-level applications and DMIF called the DMIF Application Interface (DAI). The authors defined an API instance of the DAI which conforms with the interface semantics specified in [3].

The DAI provides the required functionality for realizing multimedia applications with QoS support. Through the DAI the user may request service sessions and transport channels without concern as to the selected communication protocol. Furthermore, the DAI shields legacy applications from new delivery technologies since it is the responsibility of the underlying DMIF system to adapt to these new transport mechanisms. The DAI API specification is currently under consideration as a novel contribution to the MPEG-4 standard.

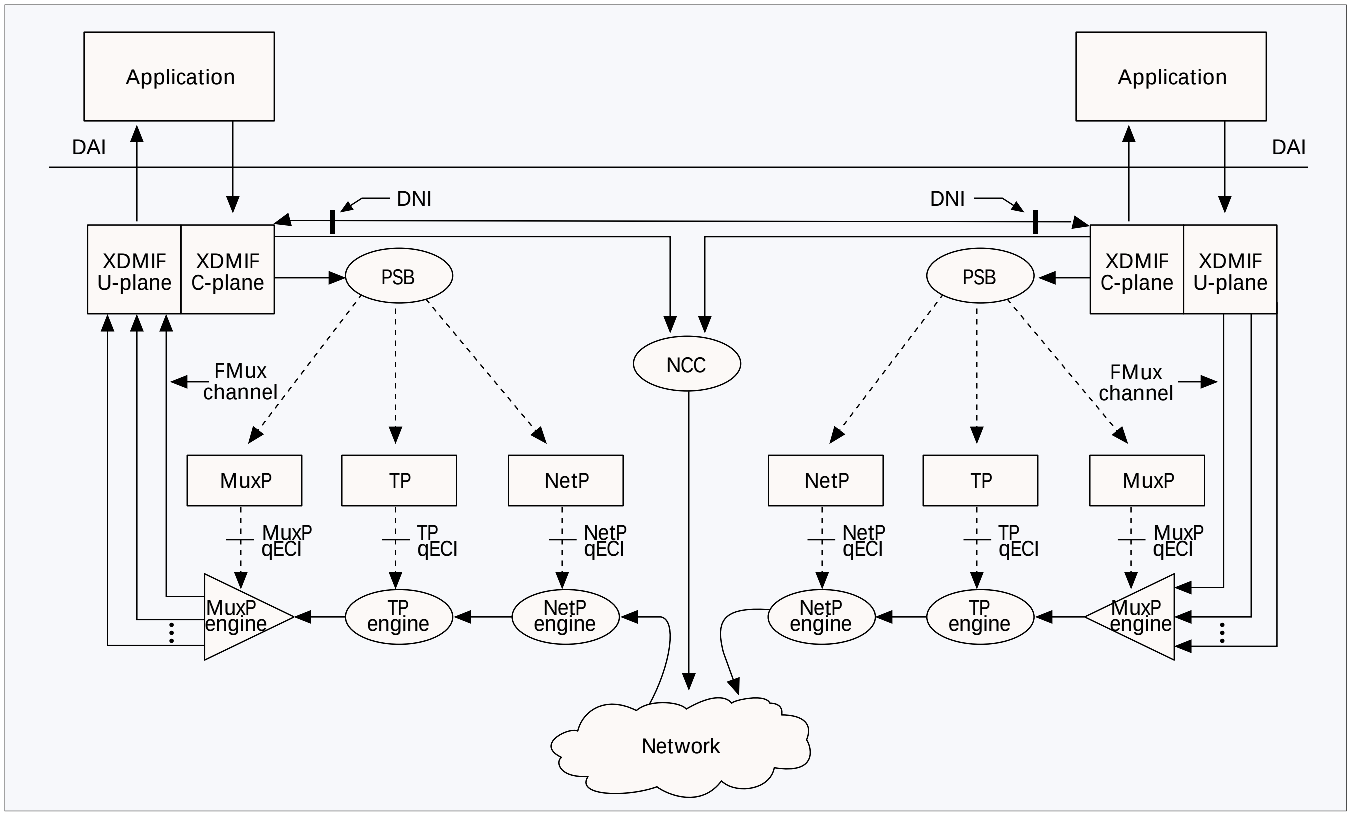
XDMIF

Figure 6 depicts the system architecture of *XDMIF*, a realization of DMIF within the framework of *xbind* [5], a broadband kernel for multimedia networks, and illustrates some of the interactions and components involved during the creation of a multimedia service.

The *xbind* broadband kernel is an open programmable networking platform for creating, deploying, and managing sophisticated next-generation multimedia services. It is conceptually based on the XRM — a reference model for multimedia networks [2]. The term *broadband kernel* is deliberately used to draw an analogy to the operating-system-like functionality the system must support, namely, that of a resource allocator and an extended machine.

The architecture of *XDMIF* consists of a transport plane and a control plane; these are denoted in Fig. 6 as U-plane and C-plane, respectively. The *XDMIF* C-plane functionality is implemented around the DMIF network interface (DNI); for a detailed description of the DNI see [3]. The transport architecture of *XDMIF* allows for the dynamic creation of protocol stacks by binding transport components at runtime. It further allows multiple implementations of a protocol stack layer to coexist within a single application and for new capabilities to be added without having to modify any of the existing architectural components. In this way, the same applications can interoperate across ATM, Internet, mobile, and telephone networks.

The MuxP engine provides the capability of multiplexing



■ **Figure 6.** *The XDMIF system architecture.*

data with similar QoS requirements into one transport channel. It implements the transport capabilities of the DMIF FlexMux, the multiplexer of MPEG-4 elementary streams. The TransMux layer of DMIF is mapped into the combination of TP and TP engine and NetP and NetP engine pairs. These can be instantiated, for example, for interworking with ATM Forum or IP-based networks.

xbind provides the QoS framework needed by DMIF for QoS management. In particular, the PSB ensures that MuxP has the needed resources available in its transport channel before establishing a new DMIF channel. If, in order to satisfy a new channel request, an insufficient amount of resources are available at the FlexMux layer, the *XDMIF* C-plane requests a new network channel from the network connection controller (NCC). Subsequently, the PSB constructs a new stack (NetP, TP, and MuxP). Hence, *xbind* provides QoS management capabilities to *XDMIF* in a natural way.

The transport architecture described in this article has been used throughout the entire *XDMIF* implementation. The only new transport component implemented was the FlexMux engine, the special DMIF requirement for supporting the multiplexing of MPEG-4 elementary streams.

Related Work

Programmable TP stacks have been the subject of a considerable amount of research in the past. Related work can be found under the topics of flexible protocol stacks, adaptive protocol stacks, universal asynchronous protocol interfaces, object-oriented transport components, configurable protocol stacks, and so on [6–9]; all address the issue of programmable transport, but do not consider issues of QoS. Reference [10, 11] addresses both QoS and programmability issues; [12] presents the x-kernel, an operating system environment that provides an explicit architecture for constructing and composing network protocols. Reference [13] also provides a taxonomy of key transport system services and illustrates the concepts with a survey of four operating system transport architectures, namely, System V and BSD UNIX, the x- kernel, and Choices. Finally, [14] addresses many of the issues related to the implementation of protocol stacks at the user level.

In [6] a protocol environment populated by standard communication functions is proposed. Applications have the ability to compose protocol stacks out of the standard protocol entities by interconnecting them. A generic communication bootstrap procedure is proposed, where new protocol components can be downloaded and installed at runtime. The approach proposed in [6] is similar to ours and deviates from the strict layering model imposed by standards such as the Open Systems Internconnection (OSI) protocol suite or UDP/TCP/IP protocols.

In [7] a universal protocol interface that allows the initiator of a communication session to define in detail the rules of information exchange is presented. The universal asynchronous protocol interface is a data link device that can download (arbitrary) protocols and execute them. It is also proposed that data transfer protocol logic be dynamically switched at runtime. Finally, protocols must be specified using a general instruction set that looks very much like an assembler to which instructions specific to communication protocols have been added.

In [8] a dynamic protocol configuration capability based on three types of protocol elements is proposed. The protocol stacks' bottom layer is called the *device element* and is similar to NetP. It abstracts a particular hardware device or kernel interface that it manages. The *endpoint element* defines the API to the protocol stack. Finally, the *protocol elements* are the intermediate objects in the protocol stack. They provide transmission and reception interfaces to their neighbors. The protocol element abstracts the various protocol stack functions (e.g., multiplexing) and specific protocols such as RPC. In this framework, TP and MuxP are protocol elements. The binding plane allows the dynamic configuration of protocols by specifying protocol stacks via a reference structure that the server can provide to clients upon request.

In [9] a Java-based system with dynamic protocol stack building capability is presented. The protocol stack elements are Java-based and inherit the protocol stack structure from a base protocol class. The protocol structure defines methods for binding "higher-" and "lower-" layer protocols. The protocol stack is constructed dynamically at runtime by a special service class that allocates, initializes, and interconnects the various protocol elements.

In [10] an adaptive transport system that can be dynamically configured to meet application needs is presented. The system allows negotiation of policies and transport mechanisms with remote hosts and intermediate service nodes to determine the protocol stack configuration that would meet the QoS requirements of an application. A factory of protocol elements (kernel objects) instantiates and interconnects the protocol elements to create useful protocol stacks. The work presented in [10] puts special emphasis on the adaptation capability of the system to reconfigure the transport system to meet the application QoS requirements after drastic changes have taken place in the network.

In [11] a function-based communication model that allows applications to request tailored services from the communication subsystem is presented. The protocol objects implement a single specific functionality such as sequence control, flow control, error correction, segmentation, sequencing, and so on. These services are known as *protocol functions*. Using the available protocol functions, the protocol configurator creates protocol finite state machines based on the application service requirements and available resources. The approach in [11] abandons the concept of layering and has finer control granularity than the approach presented in this article. Our architecture proposes to use engine protocol objects that fully implement protocol state machines with all of their functionality (e.g., TCP, XDR). In our approach, protocol stacks are dynamically, and possibly remotely, configured. In [11] the protocol state machines are dynamically (and locally) configured based on application needs.

In [12] the x-kernel is presented. The x-kernel is an operating system with special kernel support for network protocols (e.g., memory map, buffer management, and event managers). Initially, protocol stacks are statically specified using a graph when the kernel is configured. When the kernel is booted, each protocol object is initialized and waits to receive messages. At runtime, when an operation is invoked by an application, a message recursively traverses the graph, visiting each protocol and session object on its path until it reaches the network interface cards. Processes are associated with messages rather than protocols. This has the advantage that messages can in general be entirely processed with no context switch. Although the x-kernel is configurable, the protocol graph is not dynamic. At runtime, a message can only decide to skip a protocol object if it does not desire its service. However, the possible protocol stacks are predefined.

Traditionally, communication transport systems have resided entirely in the operating system. In our architecture as well as in [6, 8, 9], it is proposed to implement lightweight protocol elements and build the protocol stacks in user space. Reference [14] discusses some of the implications for an overall communication system structure to support efficient user-level implementation of protocols.

Finally, from the standpoint of APIs our architecture is similar in philosophy to the Microsoft Windows open software architecture for which both the APIs and service provider interfaces (SPIs) are specified. Software developers build applications using the APIs independent of service providers, who implement service components compliant with SPIs. As long as the service providers satisfy the SPI requirements, applications built using the APIs run properly.

Conclusions

In this article we describe an object-oriented transport architecture in which the atomic processing entity is based on the consumer/producer model. The architecture consists of consumer/producer components that are separately represented by their transport abstraction, their control and management abstractions, and, a set of controllers implementing network services such as the dynamic binding of protocol stacks.

From the control standpoint, the state of the consumer/producer is made available to the programmer and controllers through the binding interface base. The BIB is a collection of CORBA interfaces representing networking and end system multimedia resources [15]. The interfaces of the consumer/producer front-ends are part of the BIB. They consist of QoS-based APIs used by the binding controllers and management system to control and manage the engines. Since consumer/producer front-ends are open CORBA interfaces they allow for end system transport and computing resources to be remotely controlled.

The protocol stack is modeled in an object-oriented manner. The PSB dynamically creates a variety of protocol stacks on a per-call basis that are tailored to the special needs of the application. This differs significantly from the traditional transport architecture which assumes preinstalled TP stacks that cannot be customized.

To illustrate some of the advantages provided by the architecture, we describe the first reference implementation of the ISO MPEG-4 Delivery Multimedia Integration Framework and demonstrate how quickly this standard was implemented in our framework.

References

[1] M. C. Chan *et al.*, "On Realizing a Broadband Kernel for Multimedia Networks," *Proc. 3rd COST 237 Wksp. Multimedia Telecommun. and Apps.*, Barcelona, Spain, Nov. 25–27, 1996.

[2] A. A. Lazar, "Programming Telecommunication Networks," *IEEE Network*, Sept./Oct. 1997, pp. 8–18.

[3] ISO/IEC 14496-6 CD "Delivery Multimedia Integration Framework, DMIF," May 1998.

[4] J.-F. Huard and A. A. Lazar, "On QoS Mapping in Multimedia Networks," *Proc. 21st IEEE Annual Int'l. Comp. Software and App. Conf. (COMPSAC '97)*, Washington, DC, Aug. 13–15, 1997.

[5] Project xbind at Columbia University: http://comet.columbia.edu/xbind, Summer 1996.

[6] C. Tschudin, "Flexible Protocol Stacks," *Proc. ACM SIGCOMM '91*, Zurich, Switzerland, 1991, pp. 197–204.

[7] G. Holzmann, "Standardized Protocol Interfaces," *Software — Practice and Experience*, vol. 23, no. 7, July 1993, pp. 711–93.

[8] J. S. Crane, "Dynamic Binding for Distributed Systems," Ph.D. thesis, Univ. of London, 1997.

[9] B. Krupczak, K. L. Calvert and M. Ammar, "Implementing Protocols in Java: The Price of Portability," *Proc. IEEE INFOCOM*, Mar. 1998.

[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," *Proc. Symp. High Perf. Dist. Comp.*, Syracuse, NY, Sept. 1992, pp. 174–86.

[11] M. Zitterbart, B. Stiller, and A. N. Tantawy, "A Model for Flexible High-Performance Communication Subsystems," *IEEE JSAC*, vol. 11, no.4, May 1993, pp. 507–18.

[12] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An Architecture for Implementing Network Protocols," *IEEE Trans. Software Eng.*, vol. 17, no. 1, Jan. 1991, pp. 64–76.

[13] D. C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communications Systems," *IEEE JSAC*, vol. 11, no.4, May 1993, pp. 489–506.

[14] C. A. Thekkath *et al.*, "Implementing Network Protocols at User Level," *IEEE/ACM Trans. Networking*, Oct. 1993; also in *Proc. ACM SIGCOMM '93*.

[15] IEEE P1520, "Programmable Interfaces for Networks," http://www.ieee-pin.org.

Biographies

Jean-François Huard (http://www.xbind.com/~jfhuard) received a B.Eng. (EE) in 1990 from Ecole Polytechnique de Montreal, an M.A.Sc. (EE) in 1992 from Concordia University, Montreal, and an M.Phil. in 1994 from Columbia University, New York. He his currently a Ph.D. candidate in the Department of Electrical Engineering at Columbia University. His current research interests are in the area of open middleware transport architecture and high-performance QoS-aware transport protocols. From 1992 to 1998 he was a graduate research assistant with the COMET Group in the Center for Telecommunications Research at Columbia University. During the summers of 1994 and 1995 he was with AT&T Bell Laboratories, Murray Hill, New Jersey. He was awarded a Centennial Scholarship by the NSERC of Canada (1990–1994). Currently he is leader of the Transport Group of Xbind, Inc. He is also technical editor for the IEEE P1520 standard initiative and chair of the IEEE P1520 ATM SWG (http://www.ieee-pin.org).

Aurel A. Lazar [F '93] (http://comet.columbia.edu/~aurel) has been a professor of electrical engineering at Columbia University since 1988. His current theoretical research interests are on networking games and pricing. His experimental work focuses on building open programmable networks. This work has led to the establishment of the IEEE Standards Working Group on Programming Interfaces for Networks (http://www.ieee-pin.org). He was instrumental in establishing the OPENSIG (http://comet.columbia.edu/opensig) international working group with the goal of exploring network programmability and next-generation signaling technology. He was program chair of the Fall and Spring OPENSIG '96 workshops and of the First IEEE Conference on Open Architectures and Network Programming (OPENARCH '98, http://comet.columbia.edu/openarch). Currently on leave from Columbia University, he is chairman and CEO of Xbind, Inc., a high-technology startup located in Manhattan's Silicon Alley (http://www.xbind.com).