Configuring Function-based Communication Protocols for Multimedia Applications

Douglas C. Schmidt
Tatsuya Suda

Info. and Comp. Sci. Dept.
University of California, Irvine
Irvine, California, USA[1]

Burkhard Stiller
Martina Zitterbart

Institute of Telematics
University of Karlsruhe
Karlsruhe, Germany

This paper appeared in the proceedings of the 8th IFIP International Working Conference on Upper Layer Protocols, Architectures, and Applications in Barcelona, Spain, June 1994.

Abstract

Next generation communication systems must support diverse applications operating over high-performance local, metropolitan, and wide area networks. This paper describes a framework that contains a number of resource, language, and tool components for generating customized protocols to support diverse multimedia applications running in high-performance network environments. These components help to simplify the process of generating application-tailored communication protocols by automating many development and configuration steps. A collaborative distance learning application scenario is presented to motivate and demonstrate techniques used to compose function-based protocols that are customized for particular application requirements. In addition, the structure of a protocol resource pool that contains reusable protocol function building-blocks is also examined.

1 Introduction

The communication requirements of emerging distributed applications are becoming increasingly diverse. Therefore, it is important to develop high-performance communication subsystems that efficiently and flexibly support this diversity, particularly for multimedia applications (such as scientific visualization, medical imaging, and collaborative work projects) that run on high-performance local and wide-area networks (such as FDDI and ATM-based B-ISDN). However, traditional communication models and protocols may be inadequate to support the requirements of the emerging applications. For example, excessive layering in communication models such as the OSI reference model results in redundant functionality and limits the potential for processing protocols on parallel platforms [1].

One method for addressing the inadequacies of traditional communication models and protocols involves creating application-tailored protocols that execute efficiently on a variety of hardware and operating system platforms [2]. This paper describes a suite of languages, resources, and tool components that form an integrated framework to facilitate the development of application-tailored protocols. The components in this framework automate many steps involved with generating customized protocols that run in parallel on heterogeneous platforms such as message passing transputers [3] and shared memory multi-processors [4]. In general, the application-tailored protocols described in this paper share two related characteristics. First, they are based on a de-layered communication model, rather than a conventional layered model [1]. Second, they are composed of reusable protocol function building-blocks (such as acknowledgement, retransmission, segmentation, reassembly, and sequencing). These functions serve as resources that may be flexibly combined to generate efficient, de-layered protocols.

The special-purpose language components presented in this paper describe and manipulate the protocol function resources in a systematic, flexible manner. The tool components [5] use the protocol function resources and languages to automatically transform platform-independent descriptions of protocol functionality into executable protocol machines that are optimized for a specific target platform. The target platforms supported by this framework differ in terms of operating system and hardware aspects such as the number of available processing elements, interprocess communication mechanisms, memory and bus architectures, and the network interface devices. However, many of the same resources, languages, tools, and underlying architectural principles may be applied on the different platforms.

The paper is organized in the following manner. Section 2 gives an overview of the function-based communication model and defines the terminology used in this paper. Section 3 presents a collaborative distance learning application that motivates and demonstrates the use of a function-

[1]Portions of the University of California, Irvine material is based upon work supported by the National Science Foundation under Grant No. NCR-8907909. This research is also supported in part by grants from the University of California MICRO program, Nippon Steel Information and Communication Systems Inc. (ENICOM), Hitachi Ltd., Hitachi America, and Tokyo Electric Power Company.

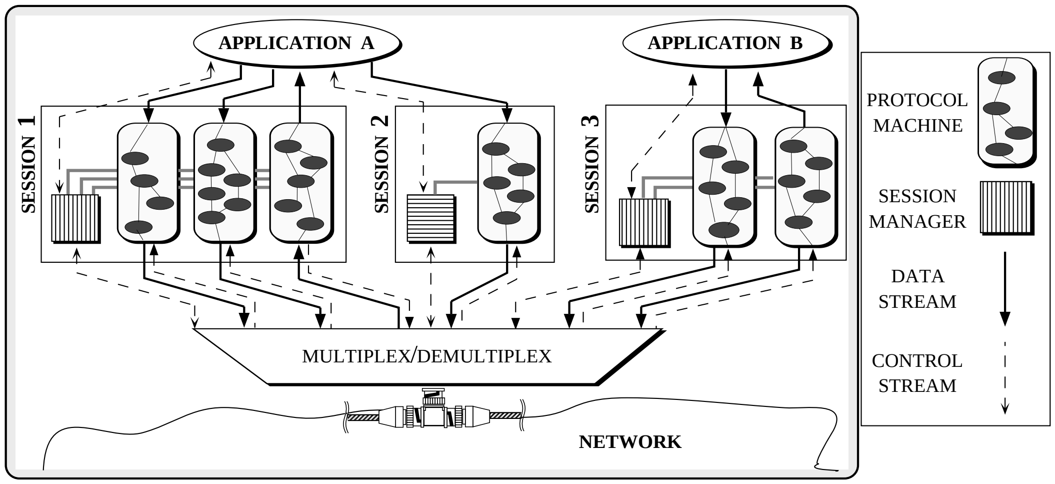


Figure 1: Components in the Function-based Communication Model

based approach for generating protocols that are customized for different types of multimedia traffic. Section 4 illustrates examples of a flowgraph-based language that describes each customized protocol used in the application scenario. Section 5 outlines a set of resources and tools that automate the generation of executable protocols described via the languages presented in earlier sections. Section 6 summarizes the paper and presents concluding remarks.

2 Overview of the Function-based Communication Model

The languages described in this paper are based upon the function-based communication model described in [1]. In this model, conventional coarse-grain hierarchical protocol layers are replaced by a communication subsystem that is decomposed into finer-grain *protocol functions*. Typical examples of protocol functions include flow control, error control, acknowledgment, and connection establishment. Each protocol function may be implemented via alternative *protocol mechanisms*. For instance, the flow control function may be implemented by either window-based and/or rate-based mechanisms.

Protocol functions serve as building-blocks for various architectural components that support the function-based communication model. For example, a particular set of functions may be combined to form a *protocol machine*. Each protocol machine is customized to support an application *data stream*. A data stream represents a uni-directional flow of application data between one or more communicating end-systems. To increase the potential for parallel processing, the sender and receiver portions of a protocol machine are decoupled as much as possible in the end-systems [6].

To support complex multimedia applications (such as collaborative distance learning or teleconferencing), multiple data streams may be consolidated to form a *session*. A *session manager* coordinates the protocol machines for related data streams within each session. For instance, a session manager handles session control information (such as synchronization points for synchronized audio and video data streams) and performs various management tasks such as adding, modifying, or deleting data streams dynamically. Each data stream in a session is implemented by a separate protocol machine that is customized for a specific set of application requirements during a particular time period. Furthermore, a protocol machine may be updated at run-time to adapt to changes in the application, local and remote operating systems, or underlying network [7].

Figure 1 depicts the relationships between the various components in the function-based communication model outlined above. In this figure, Application A maintains two sessions. Session 1 contains two outgoing data streams and one incoming data stream and Session 2 contains a single outgoing data stream. Each stream is implemented by a different protocol machine, and all protocol machines in a session are coordinated by a session manager. The network interface component is responsible for demultiplexing incoming application data onto a particular protocol machine associated with a unique session/data stream combination. This strategy enables the selected protocol machine to process data without requiring additional demultiplexing operations. This "non-layered" multiplexing approach [8, 9] reduces jitter, enables the provision of quality-of-service guarantees on a per-stream basis, and enhances parallel execution performance by minimizing synchronization overhead.

In general, the motivations for utilizing a function-based approach are to enhance service flexibility and to improve communication subsystem performance in order to better satisfy application requirements. Applications may precisely specify their quantitative and qualitative requirements via a flexible service interface described in [1]. This interface enables the communication subsystem to select or generate customized protocol machines that use protocol functions as their basic building-blocks. These protocol machines are specially-tailored to contain the minimal set of

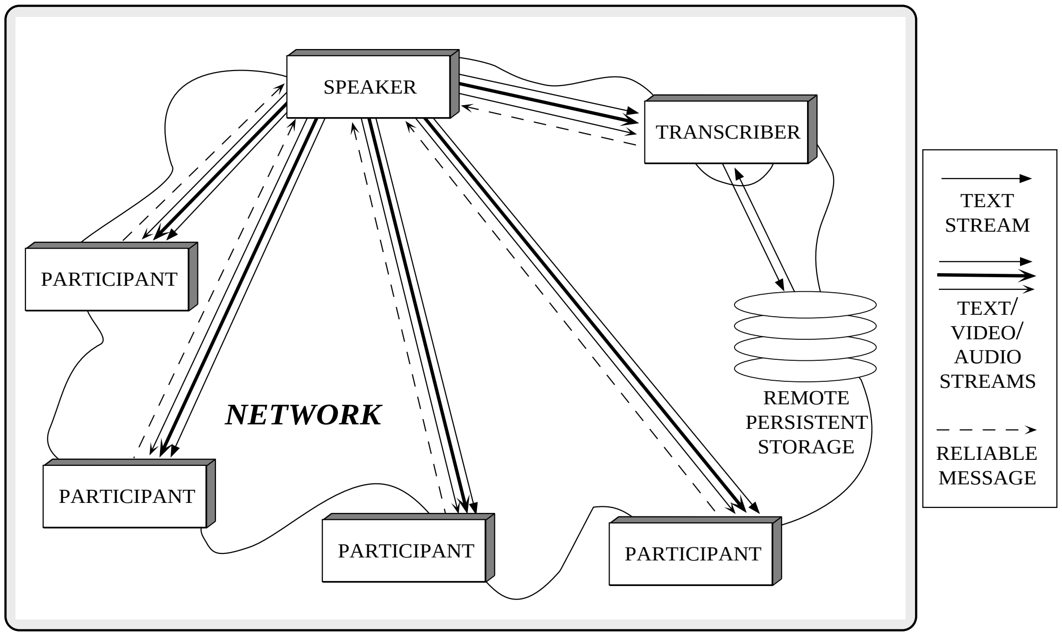


Figure 2: Topology of the Collaborative Distance Learning Application

functions required to perform a particular service. In addition, protocol functions form a convenient level of abstraction that is amenable to parallel execution on various multiprocessor platforms. Performance measurements indicate that this function-based communication model is a promising approach for developing high-performance transport systems [3].

3 Collaborative Distance Learning Scenario

This section describes the functionality of the key architectural components in a "collaborative distance learning" application. This application is used throughout the remainder of the paper to motivate and demonstrate the languages, resources, and tools provided by a framework that supports the function-based communication model. In this application scenario, a designated *speaker* gives an interactive presentation to *participants*, who are distributed throughout a network (illustrated in Figure 2). The audio and video images of the speaker are distributed uni-directionally to all participants in real-time. Each participant watches and listens to the speaker via computer end-systems that possess audio, video, and textual display capabilities, as well as text input capability from a keyboard. Although participants do not have voice or video transmission devices, they may compose and submit questions via their keyboard. These questions are delivered interactively to the speaker using a textual format. Questions arriving from participants are queued and the speaker is notified. When the speaker selects a question to answer, the question is displayed in a small window on the console of each participant. In addition, one or more of the participants may transcribe the lecture interactively, storing the text in a remote persistent storage device (such as a network file server) for subsequent retrieval.

This scenario makes several assumptions about the network environment where the application runs. For example, it is assumed that an unreliable multicast service is provided by the underlying network (such as FDDI or FDDI-II). This service supports the audio and video data streams, which are multicast by the speaker's end-system to all participants. From the perspective of the application and the protocol machines, the network multicasting service is accessible by simply supplying a multicast group address with a transmitted message. Under other circumstances, however, multicasting may require additional support from the transport system. For instance, multicast is not performed by the underlying network in certain environments (such as a wide-area B-ISDN network). In this case, the protocol machines that implement audio and video transmissions must provide multicast functionality explicitly. In addition, the example assumes that the underlying network is capable of providing certain quality-of-service guarantees for the bandwidth and error rates necessary to support the audio and video streams effectively.

Figure 3 depicts a "snapshot" of the collaborative distance learning application in operation. This figure illustrates the protocol machines in the end-systems of the speaker and one participant (the end-systems of other participants are configured similarly). The speaker possesses a multimedia-capable end-system equipped with a camera, a microphone, a display console, and a keyboard. Each data stream in the communication subsystem of an end-system is implemented via a separate protocol machine. The protocol machines are coordinated by a session manager that synchronizes the audio

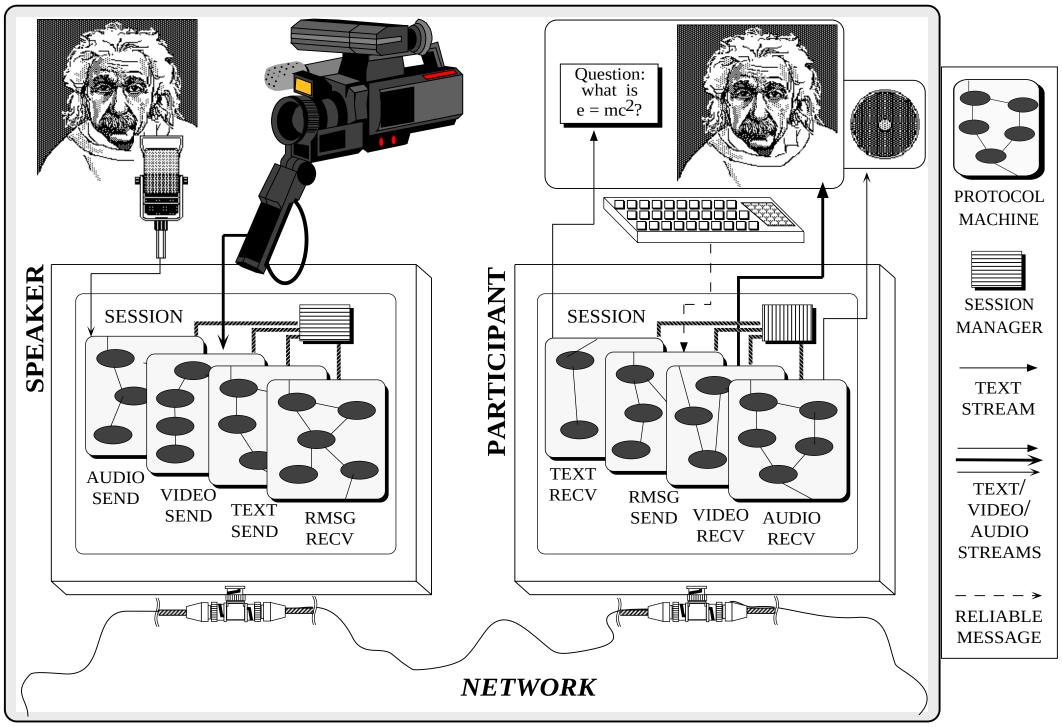


Figure 3: Dynamic Snapshot of the Communication Subsystem

and video streams. In addition, the speaker's session contains a connectionless, reliable message protocol machine for receiving questions from individual participants. Conversely, the text, video, and audio protocol machines on the speaker's end-system maintain connections to all participants. Connections are used in the network and/or the transport system of the end-systems to ensure that the necessary quality-of-service levels are provided.

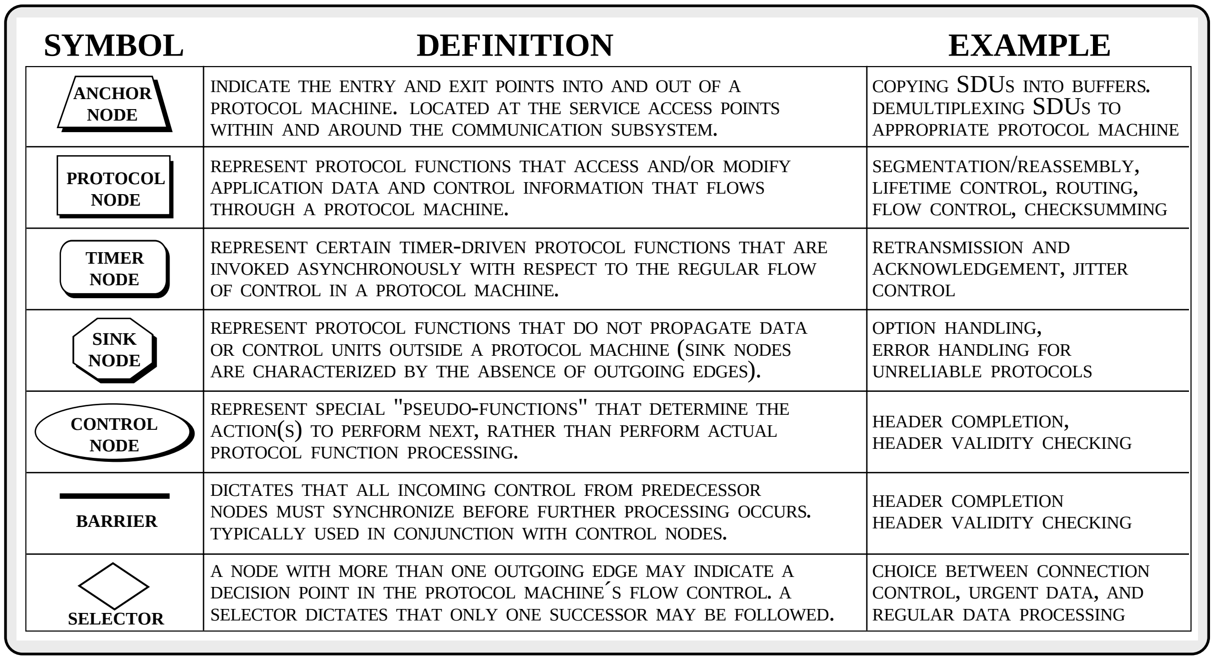
The information presented by the speaker is captured and transmitted in real-time to participant end-systems, where the video and text information is simultaneously displayed on a console and the audio is played over a loudspeaker. The participant end-systems also maintain separate protocol machines for receiving audio, video, and text. In addition, the participants may ask questions that are sent to the speaker via a reliable message protocol machine. The text stream on the speaker's end-system multicasts these questions to all participants for the duration of the speaker's answer.

Each protocol machine is implemented in a de-layered manner corresponding to the principles of the function-based communication model. De-layering involves combining the functionality of the network layer and the transport layer within each protocol machine. The primary motivations for adopting a de-layered approach are to eliminate redundant and/or extraneous functionality in the protocol machines, as well as to increase the opportunity for processing the protocol machines concurrently on parallel platforms. The characteristics of each protocol machine mentioned above are discussed further in the following section.

4 Application-Tailored Protocol Machines

This section illustrates and examines the protocol machines for the audio and video used in the collaborative distance learning application (the text and reliable message streams are omitted to save space). These data streams exchange different types of information using the network and end-system environment described in the scenario from Section 3. Each stream is implemented via a separate protocol machine that is customized for the quantitative and qualitative requirements of the data it transmits or receives. Quantitative requirements involve criteria that may be evaluated in terms of measures such as "bits per-second throughput" or "the number of bit errors tolerated per-PDU." Qualitative requirements specify the services related to session management, stream management, and data unit management [1], and may be described in terms of nominal attributes such as "inter-stream synchronization," and "in-sequence and/or unduplicated delivery of data."

The following subsections present the quantitative [10, 11, 12] and qualitative [1] requirements for several of the data streams in the example scenario. In addition, the protocol functions and mechanisms used by the audio and video protocol machines are also described (the text and reliable message protocol machines are omitted due to space limitations). The figures depict the sending and receiving protocols machines separately since each protocol machine implements a uni-directional data stream. In addition to transmitting data, however, a uni-directional sender must also be capable of re-



SYMBOL	DEFINITION	EXAMPLE
ANCHOR NODE	INDICATE THE ENTRY AND EXIT POINTS INTO AND OUT OF A PROTOCOL MACHINE. LOCATED AT THE SERVICE ACCESS POINTS WITHIN AND AROUND THE COMMUNICATION SUBSYSTEM.	COPYING SDUs INTO BUFFERS. DEMULTIPLEXING SDUs TO APPROPRIATE PROTOCOL MACHINE
PROTOCOL NODE	REPRESENT PROTOCOL FUNCTIONS THAT ACCESS AND/OR MODIFY APPLICATION DATA AND CONTROL INFORMATION THAT FLOWS THROUGH A PROTOCOL MACHINE.	SEGMENTATION/REASSEMBLY, LIFETIME CONTROL, ROUTING, FLOW CONTROL, CHECKSUMMING
TIMER NODE	REPRESENT CERTAIN TIMER-DRIVEN PROTOCOL FUNCTIONS THAT ARE INVOKED ASYNCHRONOUSLY WITH RESPECT TO THE REGULAR FLOW OF CONTROL IN A PROTOCOL MACHINE.	RETRANSMISSION AND ACKNOWLEDGEMENT, JITTER CONTROL
SINK NODE	REPRESENT PROTOCOL FUNCTIONS THAT DO NOT PROPAGATE DATA OR CONTROL UNITS OUTSIDE A PROTOCOL MACHINE (SINK NODES ARE CHARACTERIZED BY THE ABSENCE OF OUTGOING EDGES).	OPTION HANDLING, ERROR HANDLING FOR UNRELIABLE PROTOCOLS
CONTROL NODE	REPRESENT SPECIAL "PSEUDO-FUNCTIONS" THAT DETERMINE THE ACTION(S) TO PERFORM NEXT, RATHER THAN PERFORM ACTUAL PROTOCOL FUNCTION PROCESSING.	HEADER COMPLETION, HEADER VALIDITY CHECKING
BARRIER	DICTATES THAT ALL INCOMING CONTROL FROM PREDECESSOR NODES MUST SYNCHRONIZE BEFORE FURTHER PROCESSING OCCURS. TYPICALLY USED IN CONJUNCTION WITH CONTROL NODES.	HEADER COMPLETION HEADER VALIDITY CHECKING
SELECTOR	A NODE WITH MORE THAN ONE OUTGOING EDGE MAY INDICATE A DECISION POINT IN THE PROTOCOL MACHINE´S FLOW CONTROL. A SELECTOR DICTATES THAT ONLY ONE SUCCESSOR MAY BE FOLLOWED.	CHOICE BETWEEN CONNECTION CONTROL, URGENT DATA, AND REGULAR DATA PROCESSING

Figure 4: Symbols in the Flowgraph-based Protocol Machine Configuration Language

ceiving control information (such as acknowledgements for connection establishment and termination requests) that is fed back from the peer protocol machine(s).

The protocol machines appearing in figures throughout this section are portrayed via the flowgraph-based configuration language summarized in Figure 4 and described in detail in [2]. This language concisely describes the protocol function building-blocks in each protocol machine and depicts their interrelationships. The flowgraph language also indicates opportunities for exploiting parallelism between protocol functions within a protocol machine. Related examples of function-based techniques for decomposing and parallelizing existing protocols such as TCP are described in [13, 2]. Other related work addresses issues such as architectures that support function-based protocol decomposition [14, 15] and graph- and shape-based protocol configuration techniques [16, 17]. To increase clarity, the protocol mechanisms that implement each function are omitted in the figures. However, these mechanisms are discussed in the text.

4.1 Audio Protocol Machines

An audio stream is used to transmit isochronous voice traffic from the speaker to the participants. It is implemented via separate sender and receiver protocol machines containing protocol functions that support the quantitative and qualitative requirements of audio traffic. The quantitative requirements for an audio stream depend on the level of quality requested by the application. For example, if regular telephone quality audio is desired then 64 kBit/s is sufficient. However, if CD-quality audio is desired, each audio stream may require approximately 1.4 MBit/s of bandwidth (certain

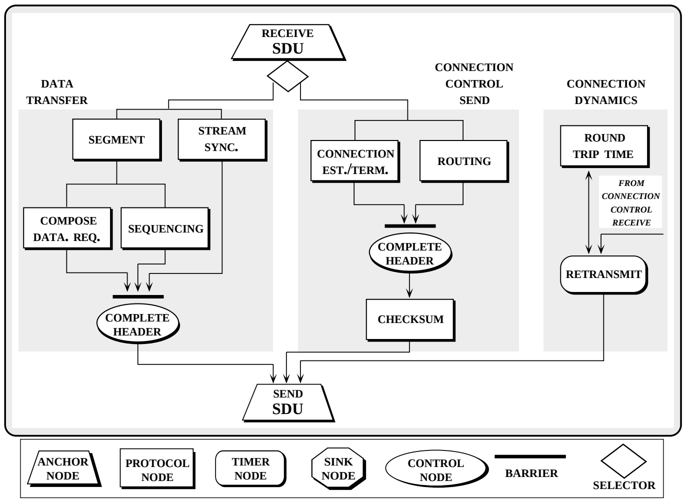


Figure 5: Protocol Machine for Audio or Video Sender

compression techniques enable the reception of high-quality audio using a 64 kBit/s channel). Bit error rates of 10^{-4} or PDU error rates of 10^{-7} are tolerable if no compression techniques are used; these error rates become more stringent if compression is applied. In addition, the inter-PDU jitter in an audio stream should not exceed 10 ms. The qualitative requirements for an audio stream include multicast transmission, with in-sequence delivery of data at the receiver, synchronization with the associated video stream, and a small, fixed-sized maximum data unit.

Figure 5 illustrates a protocol machine that implements the sender portion of an audio stream. The protocol machine is connection-oriented to expedite the detection and removal

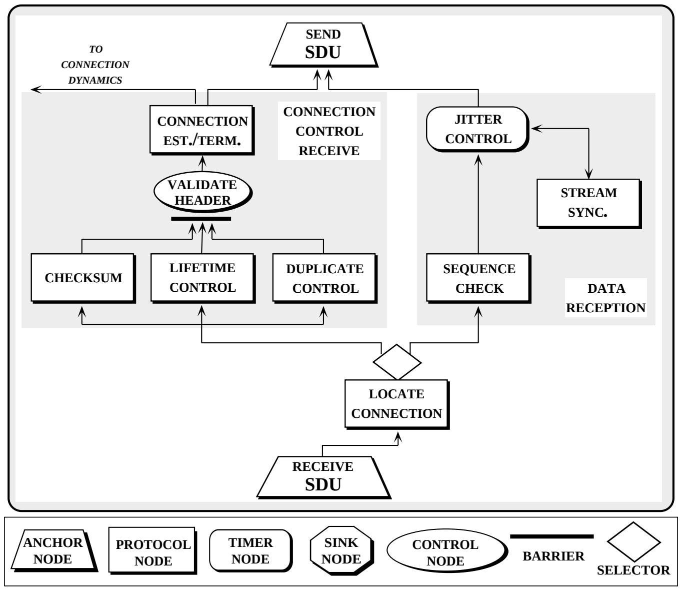


Figure 6: Protocol Machine for Audio or Video Receiver

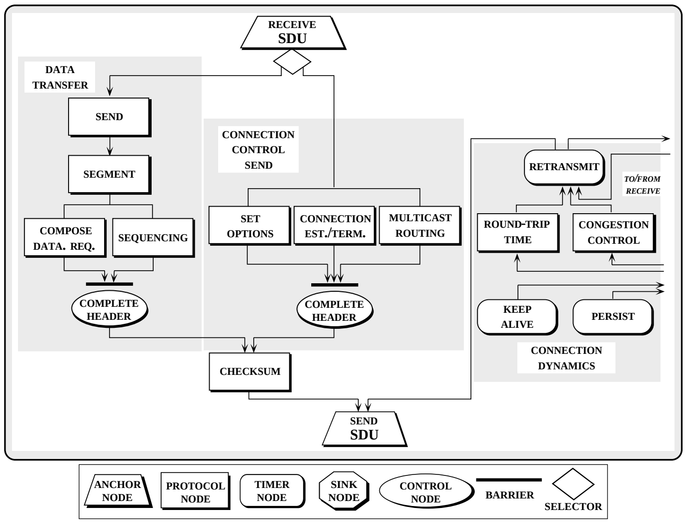


Figure 7: Protocol Machine for Text Sender

of duplicates and out-of-sequence data units. The CONNECTION ESTABLISHMENT AND TERMINATION function handles connection management issues (such as the retransmission of a connection request if no acknowledgement is received after a certain period of time). The routing identifier for each protocol machine is selected when a connection is first established. Thereafter, the entire network/host/connection address of participants need not be included with subsequent data transmissions.

The RETRANSMIT and CHECKSUM calculation functions are used in the connection control and connection dynamics portions of the protocol machine to implement reliable end-to-end connection establishment and termination semantics. Due to the isochronous, loss tolerant properties of audio data, however, the data transfer portion of the protocol machine provides neither retransmission nor checksumming of audio protocol data units (PDUs). However, a checksum of the PDU header *is* calculated to prevent the accidental acceptance of corrupted PDUs at the receiver(s). In addition, certain network properties may be used to further refine the protocol machine during connection establishment. For example, segmentation must be performed if the maximum transmission unit (MTU) size of the network is smaller than the maximum size of the application's audio service data units (SDU). On the other hand, if the MTU is larger than the maximum SDU (and assuming that the route is fixed), the SEGMENT function may be eliminated from the protocol machine entirely at connection establishment time [2].

Each SDU is assigned a unique sequence number. Moreover, if segmentation is necessary, each segment is also numbered to enable reassembly at the receiver. The contents of the data unit header are completed by storing the sequence number and data request PDU template (which contains information such as the PDU-type, the route, and the destination connection identifier which are assumed to be fixed for each PDU after connection establishment) together with the associated stream synchronization information. The STREAM SYNCHRONIZATION function calculates this information (such as playout times, maximum retrieval times, or inter-channel synchronization points [18]) to synchronize the separate audio and video data streams. This synchronization information is inserted into each PDU header and extracted by the receiver. This example assumes that audio transmission is either uncompressed or a constant bit rate compression technique is used. Therefore, SDUs are always fixed-size and the synchronization information may be added directly into SDU headers. Finally, completed data units or connection requests are transmitted via the SEND SDU function to the network interface.

The receiver-side of the audio protocol machine is shown in Figure 6. Once the appropriate connection record is located, the receiver invokes either the connection control reception functions or the data reception functions. In the connection control case, the receiver performs the CHECKSUM, the LIFETIME CONTROL, and DUPLICATE CONTROL functions (potentially in parallel) on incoming PDUs. These functions are necessary to ensure the reliability of connection establishment and termination. On the other hand, the data path of the receiver's protocol machine is streamlined since it represents the critical processing path. For example, incoming PDUs need not be acknowledged, nor are checksum calculations performed on the data portion of the PDU. The SEQUENCE CHECK function is performed since out-of-sequence data may require queueing. In particular, only in-sequence SDUs are passed to the JITTER CONTROL function, which then consults the receiver's STREAM SYNCHRONIZATION function. This function examines the appropriate fields in the PDU header and passes this information to the session manager in the receiver's session. The session manager ensures that the audio stream is synchronized with the corresponding video stream. PDUs are discarded if their useful playing time has expired. For example, if out-of-date SDUs arrive at the receiver, either the JITTER CONTROL function will not deliver them to the application or a default SDU containing "white-noise" will be inserted into the flow of SDUs. Either approach may result in a small, but acceptable, level of distortion in the audio stream.

4.2 Video Protocol Machines

The video stream periodically transmits visual samples of the speaker to participants throughout the network. As with audio, the quantitative requirements for video depend on the selected compression techniques. For example, the bandwidth requirements of an uncompressed video stream are approximately 150 MBit/s. If variable compression techniques are used, however, this amount may be reduced to approximately 32 MBit/s. Reliability requirements also depend on the compression technique. Bit error rates of $\leq 10^{-2}$ and PDU error rates of $\leq 10^{-3}$ must be provided by the underlying network, otherwise the necessary image quality may not be assured. However, the amount of tolerable SDU loss may be higher for video, depending on the technique selected for encoding video control information (such as the color table) and picture image information. If these two types of information are transmitted in different PDUs, the loss rate must be smaller than 10^{-9} for the control units and smaller than 10^{-3} for data units. On the other hand, transmitting the information as combined PDUs results in an acceptable loss of 10^{-5}. In addition, up to a certain threshold, duplication of PDUs does not adversely affect the video quality. The delay jitter requirements have a maximum of 10 ms. The qualitative requirements for video are also similar to audio. For example, video requires in-sequence delivery of isochronous samples that are synchronized with the associated audio stream.

The sender and receiver protocol machines that represent the protocol machine for transmitting constant bit rate compressed video stream are practically identical to the ones shown for the audio stream in Figures 5 and 6. Although the quantitative bandwidth requirements of video transmission are substantially larger than audio, these quantitative differences do not directly affect the existence or absence of the selected protocol functions. Instead, they primarily affect the mechanisms used to implement the functions. The similarity between the audio and video protocol machines enables the direct reuse of many protocol mechanisms, particularly those involving connection control and connection dynamics.

In this example, the only protocol mechanisms implemented differently in the receiver for the video stream versus the audio stream are the mechanisms for the SEQUENCE CHECK and JITTER CONTROL functions. In particular, if no SDU is available to deliver to the application (*e.g.,* if the subsequent SDU did not arrive in time or was discarded due to buffer overflow or corruption), the previously received video SDU may be passed to the application again. Although this strategy may reduce the quality of the video slightly, it is an acceptable loss recovery strategy as long as the picture

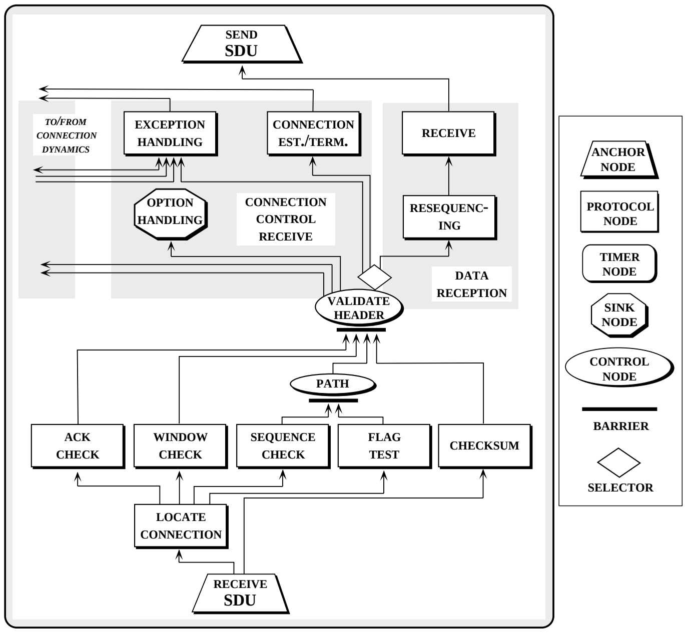


Figure 8: Protocol Machine for Text Receiver

remains comprehensible. In general, although both the video and audio protocol machines discard out-of-date SDUs, the video machine may resubmit an earlier SDU to the application, whereas the audio protocol machine submits either nothing or white-noise.

4.3 Text Protocol Machines

The text stream serves two purposes in the multimedia application scenario. First, it is used to forward participant questions received by the speaker to the other participants. Second, it implements the text stream used by participants who store and retrieve transcriptions of the speaker's presentation in the remote persistent storage repository. Most quantitative requirements of the text stream are less demanding than those for voice and video. For example, a single page of text requires approximately 20 kBit of data. Depending on the specific delay requirements, the bandwidth necessary to transmit data via the text stream may vary. For example, transmitting 20 kbit of data while maintaining a 1 sec maximum delay requires a bandwidth of 20 kBit/s. If the same amount of data is transmitted within 0.1 sec the bandwidth requirements increase to 200 kBit/s. The current application has minimal delay requirements and no jitter control is required. On the other hand, the reliability requirements are quite stringent and the text stream must provide completely reliable service (*i.e.,* zero bit or PDU errors). The qualitative requirements of the text stream necessitate in-sequence, non-duplicated delivery, a reliable multicasting facility, and potentially segmented delivery. Many existing protocols (such as TCP and TP4) provide efficient transmission of reliable, in-sequence, non-duplicated text. However, these conventional protocols do not define a standard multicasting service. Therefore, an application-tailored protocol machine for multicasting textual data is presented here. Note that there is no need to synchronize the text stream with the other data streams.

As illustrated in Figure 7, the connection-oriented protocol machine for the text sender contains three distinct components: *connection control send*, *connection dynamics*, and *data transfer*. The connection control send component is essentially identical to the ones used for the audio/video senders since it requires the same functionality. The congestion control and connection maintenance functions (such as the round-trip time estimates and the keep alive and persist timers) in the connection dynamics component are also similar to the audio/video protocol machines. The primary difference is that the text stream must handle retransmission of multicast data to participant end-systems. Note that portions of the connection dynamics component are shared by both the sender and receiver protocol machines. The data transfer component is enhanced to handle multiple options (such as extending the size of the flow control window) and therefore contains additional functions to check PDU flags and window credits.

The receiver part of the text stream protocol shown in Figure 8 involves two parts, only one of which is selected at run-time. The appropriate selection depends on the type of PDU that is received, as well as the state of the protocol machine associated with the located connection record. When a PDU is received, multiple tests (such as WINDOW, SEQUENCE, and ACK CHECK) are performed, along with the CHECKSUM calculation and the FLAG TEST (which determines whether a connection control PDU or a data PDU has arrived). These tests may be performed concurrently, depending on the level of parallelism available on an end-system. If no errors are encountered and a data PDU has arrived, the data reception portion of the receiver resequences the incoming text segments and passes them up to the application. If a connection establishment or termination PDU is received, the connection control functions are performed. Depending on the error reporting mechanism, the EXCEPTION HANDLING function may either do nothing (*e.g.,* if the sender uses a timer-based retransmission mechanism) or it may inform the sender to retransmit some or or all data if errors are detected (*e.g.,* if a selective repeat mechanism is used).

4.4 Reliable Message Protocol Machines

A reliable message stream is used to transmit questions from participants to the speaker. Since there may be many participants, a dedicated connection is *not* established in advance between each participant and the speaker, nor are direct connections maintained between the participants. This approach reduces the amount of context information that must be stored at the end-systems of the speaker and the participants [19], which enhances the scalability of the collaborative distance learning application.

The quantitative requirements of the reliable message service are similar to the text version. For instance, no PDU or bit errors are permitted since reliable service is required. The throughput requirements vary depending on the amount of data to transmit and the maximum acceptable delay. The qualitative requirements of the application are somewhat less stringent than the text stream, however. For example, multiple questions from a single participant may be delivered out-of-sequence (since the speaker is free to answer questions in any order). As with the text stream example, no synchronization is required with other streams in the scenario.

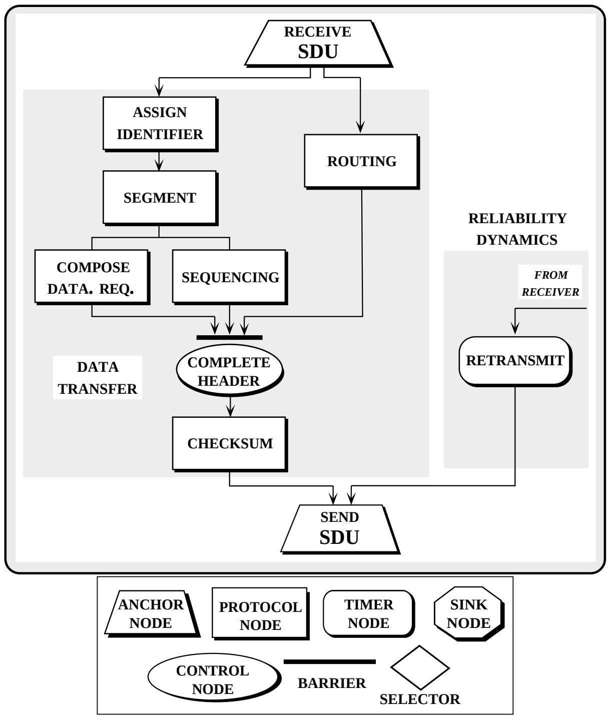


Figure 9: Protocol Machine for Reliable Message Sender

The protocol machines describing the sender and receiver portion of the reliable message service are depicted in Figure 9 and Figure 10, respectively. When a question is submitted to the protocol machine by a participant, the ASSIGN IDENTIFIER function grants an identifier to the message that is unique for the local end-system. This identifier is used to trigger acknowledgements and retransmissions with the receiver. Segmentation is required at the sender if the SDU size of a participant's question is larger than the underlying network's MTU. If segmentation is required, the SEQUENCING function is responsible for numbering the segments that are created. This information is combined with the result of the route calculation to form a data request header. The checksum of the resulting PDU is computed and the PDU is sent (note that the CHECKSUM and SEND SDU functions may be combined if the network interface computes the checksum as it is copying the PDU onto the network).

In the receiver's protocol machine, every incoming SDU is de-encapsulated and the resulting PDU is checked (potentially in parallel) for the following three conditions:

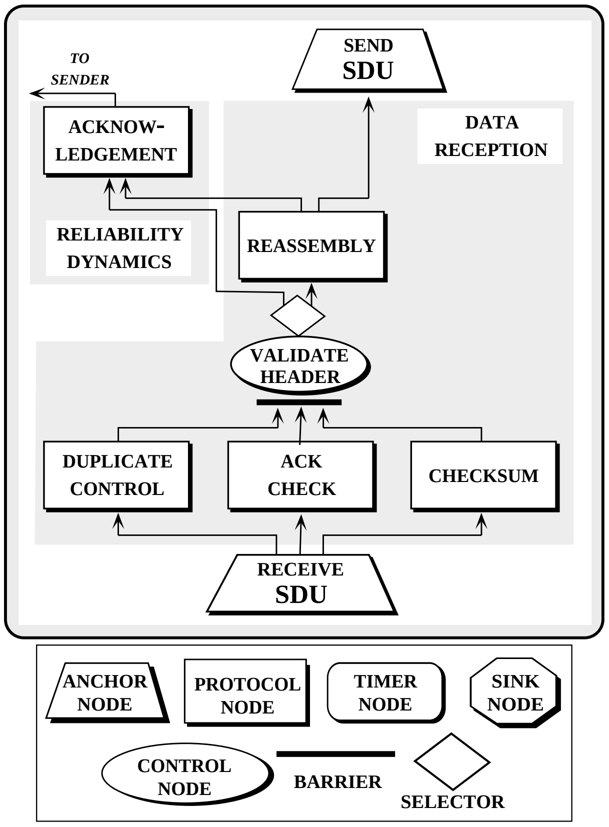


Figure 10: Protocol Machine for Reliable Message Receiver

1. Duplicate messages are detected and discarded via the DUPLICATE CONTROL function
2. The ACK CHECK function handles incoming acknowledgements that correspond to unacknowledged messages
3. The checksum of every PDU is also calculated since the message protocol machine must protect against bit errors

The results of these three functions are evaluated in the VALIDATE HEADER rendezvous node to determine whether to proceed with the acknowledgement or the reassembly of segments. Depending on the condition of the incoming PDU (as well as the underlying protocol mechanisms), the PDU will either be discarded, the ACKNOWLEDGEMENT function will request the retransmission of certain segments, or a positive ack will be transmitted to the sender.

Unlike the connection-oriented text stream, explicit connection dynamics are not used in the reliable message protocol machine since it provides a connectionless service. However, to maintain reliability and to prevent duplicated PDUs from confusing the receiver's protocol machine, an error control mechanism is employed to recover from lost or damaged messages or segments. Error control is implemented via a RETRANSMIT function that uses either a selective-repeat mechanism or a timer-based mechanism. If there are no errors, the received message will be reassembled and passed to the application via the SEND SDU function.

4.5 Comparing the Protocol Machines

It is instructive to compare the similarities and differences between the protocol machines for the four types of application requirements described above. For instance, a major difference between the sender portions of the audio/video/text protocol machines and the reliable message protocol machine is the absence of the explicit connection control in the message sender portion. Likewise, the audio and video protocol machine include mechanisms to control jitter, whereas the text and reliable message machines do not.

However, many of the protocol functions used in each protocol machine are also quite similar. Often, they are simply interconnected in a different manner and/or select different protocol mechanisms. For example, all the protocol machines use similar routing and checksum functions to ensure reliable delivery for the relevant portions of their PDUs (note that audio and video do *not* checksum the data portions of their PDUs). A comparison of the receiver portions of the protocol machines illustrates that certain functions may be used both for the connection control path and/or for the data reception path. In particular, functions that provide reliable delivery for the connection dynamics (such as retransmission and checksumming) may be added or removed from the data path as necessary.

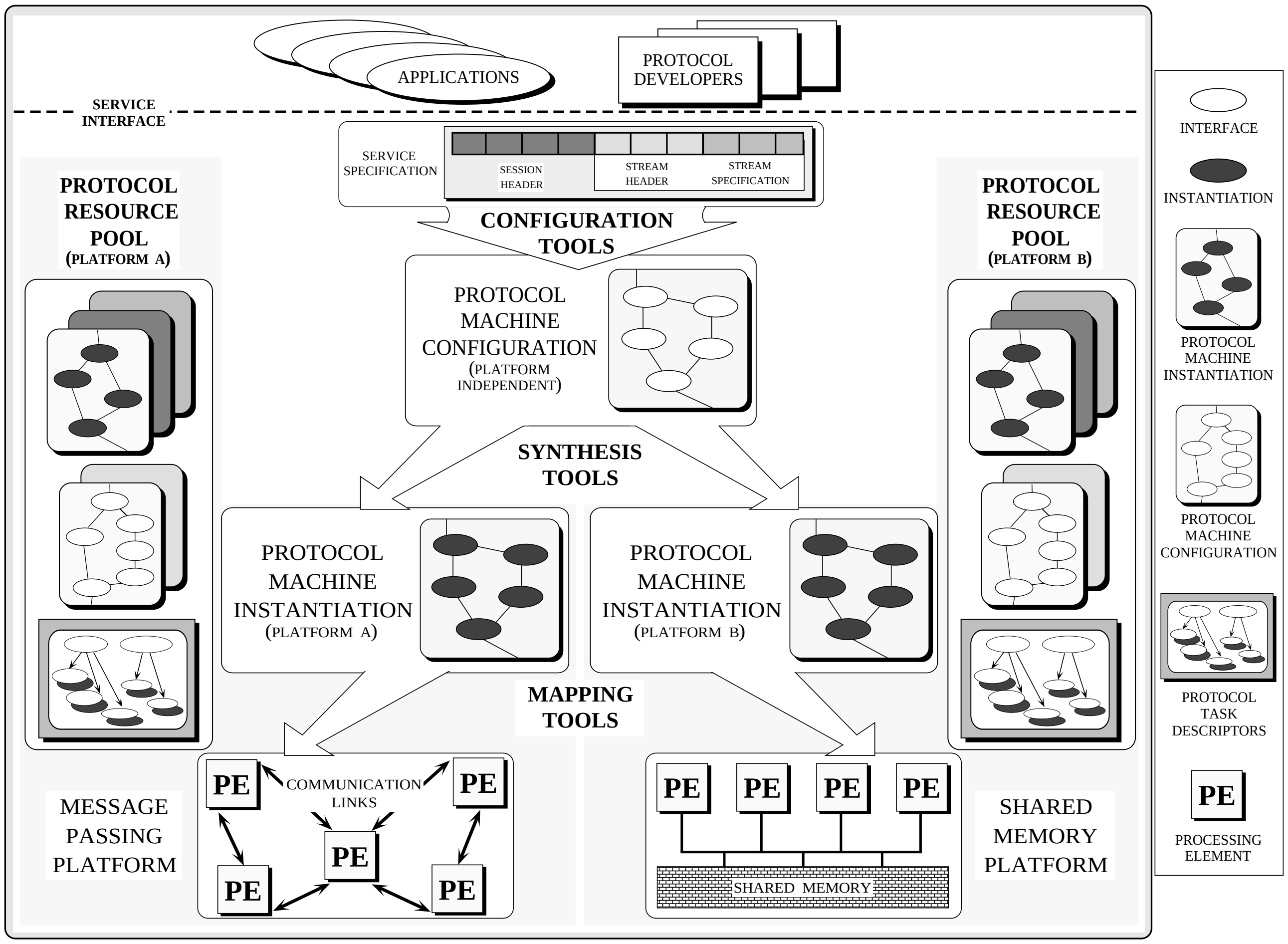
5 Resource, Language, and Tool Support

This section outlines a set of resources, languages, and tools that are used to automate the generation of executable protocol machines. In addition to simplifying the process of generating protocols by automating certain development steps, these tools also facilitate the mapping of platform-independent protocol machines onto several types of multiprocessor end-system architectures.

5.1 Classes of Tools

Figure 11 illustrates the several classes of tools used to transform high-level descriptions of qualitative and quantitative application service requirements into lower-level protocol machines that may be directly executed on a particular target platform. These tool components access and manipulate the descriptors in a protocol resource pool to transform platform-independent configurations of protocol functionality into executable protocol machine instantiations that may be optimized for a specific target platform.

Three classes of tools, *configuration*, *synthesis*, and *mapping*, are involved in configuring, instantiating, and executing application-tailored protocol machines, respectively. *Configuration tools* derive the platform-independent protocol machine configuration from the requirements specifications via a process of *selection* and *ordering*. *Synthesis tools* transform these platform-independent protocol machine configurations into executable protocol machine instantiation by



APPLICATIONS
PROTOCOL DEVELOPERS
SERVICE INTERFACE
SERVICE SPECIFICATION
SESSION HEADER
STREAM HEADER
STREAM SPECIFICATION
CONFIGURATION TOOLS
PROTOCOL RESOURCE POOL (PLATFORM A)
PROTOCOL RESOURCE POOL (PLATFORM B)
PROTOCOL MACHINE CONFIGURATION (PLATFORM INDEPENDENT)
SYNTHESIS TOOLS
PROTOCOL MACHINE INSTANTIATION (PLATFORM A)
PROTOCOL MACHINE INSTANTIATION (PLATFORM B)
MAPPING TOOLS
COMMUNICATION LINKS
PE
MESSAGE PASSING PLATFORM
SHARED MEMORY
SHARED MEMORY PLATFORM
INTERFACE
INSTANTIATION
PROTOCOL MACHINE INSTANTIATION
PROTOCOL MACHINE CONFIGURATION
PROTOCOL TASK DESCRIPTORS
PROCESSING ELEMENT

Figure 11: Resources and Tools used during Protocol Generation and Execution

composing and *interconnecting* protocol mechanism object-code and related data. *Mapping tools* transfer the instantiations into the run-time system of the target platform. The mapping process involves *local end-system resource allocation, placement*, and *loading* of particular clusters of protocol mechanisms in the instantiation onto one or more processing elements. In general, the synthesis and mapping tools perform the platform-dependent transformations, whereas the configuration tools are intended to be platform-independent. The structure and functionality of these tools is described further in [5].

5.2 The Protocol Resource Pool

Each protocol machine presented in Section 4 is composed from a set of *protocol resources*. These protocol resources are stored in a *protocol resource pool*. This resource pool is an information repository that maintains a semantically-attributed collection of *protocol resource descriptors*. These descriptors may be inserted and manipulated via a *protocol resource descriptor language* [2]. This section outlines the contents of the protocol resource pool.

There are several types of descriptors in the protocol resource pool (summarized in Figure 12). A *protocol function descriptor* represents tasks such as segmentation, reassembly, retransmission, connection establishment and termination, and flow control. Other descriptors include *control functions*, *anchor functions*, and pre-defined *protocol machine configurations* and *protocol machine instantiations*. Control functions are used for synchronizing protocol functions, as well as to determine which successor function to select at a multi-path decision point in a protocol machine flow-graph (cf. Figure 4). Anchor functions perform tasks at the boundaries of the transport system such as copying PDUs into system buffers and demultiplexing PDUs to the appropriate protocol machines. Protocol machine configurations are composite entities that contain a set of protocol resource descriptors, their predecessor and successor relations, and any synchronization information necessary to coordinate the protocol functions. A configuration is not directly executable since it only *describes* the necessary characteristics of a protocol machine. A protocol machine instantiation, on the other hand, is an executable representation of a protocol machine configuration. Instantiations contain protocol resources (such as object code and data) that may be optimized to run efficiently on a particular target platform.

The relationship between the different types of descriptors

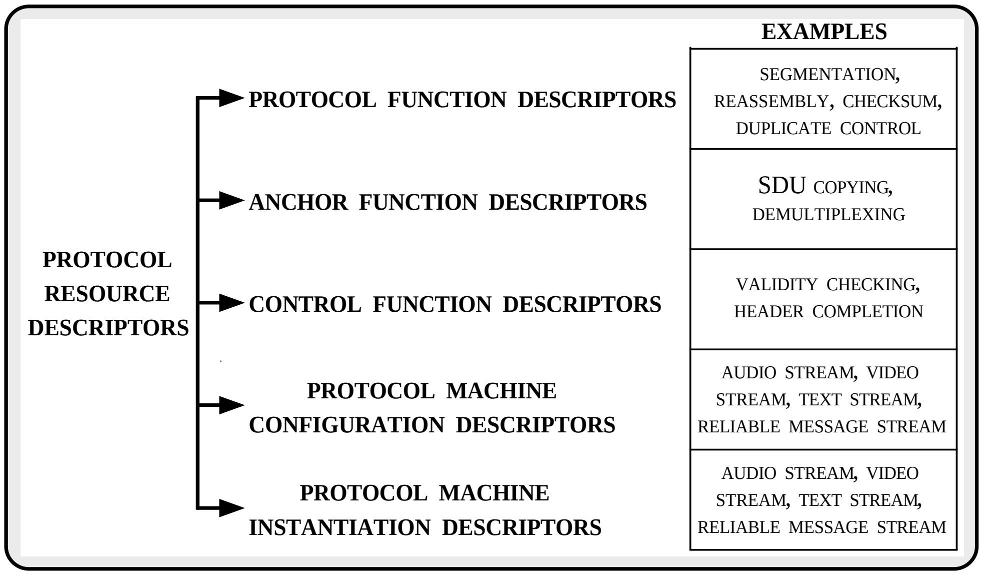

EXAMPLES
PROTOCOL RESOURCE DESCRIPTORS
PROTOCOL FUNCTION DESCRIPTORS
SEGMENTATION, REASSEMBLY, CHECKSUM, DUPLICATE CONTROL
ANCHOR FUNCTION DESCRIPTORS
SDU COPYING, DEMULTIPLEXING
CONTROL FUNCTION DESCRIPTORS
VALIDITY CHECKING, HEADER COMPLETION
PROTOCOL MACHINE CONFIGURATION DESCRIPTORS
AUDIO STREAM, VIDEO STREAM, TEXT STREAM, RELIABLE MESSAGE STREAM
PROTOCOL MACHINE INSTANTIATION DESCRIPTORS
AUDIO STREAM, VIDEO STREAM, TEXT STREAM, RELIABLE MESSAGE STREAM

Figure 12: Types of Protocol Resource Descriptors

are illustrated in Figure 12. The protocol, anchor, and control functions are located in the resource pool to facilitate various types of reuse and to simplify automated generation. The protocol machine configurations and instantiations are stored in the resource pool to reduce application start-up overhead at run-time since some or all of the time-consuming configuration and synthesis phases are omitted. This section provides several examples that indicate the type of attributes stored in the protocol resource pool and examine how various configuration and synthesis tools use the resources.

Figures 13 and 14 present an excerpt of the descriptors residing in the protocol resource pool on the speaker's end-system. Each descriptor may be defined independently and the ordering of the descriptors in the pool is insignificant. Certain attributes must be included with each descriptor. These mandatory attributes include the descriptor type, the descriptor name, the mechanism name(s) (multiple mechanisms may be included for each descriptor), the mechanism input and output parameter lists, and the associated object-code that implements each mechanism. Other attributes (such as predecessors and successors dependencies and other semantic constraints [2]) are optional and are included with a descriptor only if necessary.

Descriptors may be accessed and manipulated either manually (by protocol machine developers) or by automated configuration and synthesis tools. In essence, the collection of protocol resource descriptors shown in Figures 13 and 14 represent the "instructions" appearing in the flowgraph-based representation of the protocol machines depicted by the figures in Section 4. A short explanation of each resource descriptor in Figures 13 and 14 is given in the following subsection.

5.3 Example Protocol Resource Descriptors

As shown in Figures 13 and 14, each resource descriptor may contain one or more mechanisms. Various configuration and synthesis tools access and manipulate these mechanisms via their abstract interfaces, which include the name of each mechanism, its input and output parameters, as well as semantic information that characterizes the mechanism behavior and any constraints on mechanism use. In all these examples, the `Message` parameter refers to a composite data structure that is capable of efficient operations (such as encapsulation/de-encapsulation, segmentation/reassembly) on application and network SDUs and PDUs [20].

The first five descriptors shown in Figure 13 are used on the sender-side of a protocol machine. The protocol function `Segment` logically splits an input SDU `Message` into smaller pieces that fit within the MTU size of the underlying network. It produces a composite `Message` that contains the segments corresponding to the original SDU (the PDU header is not entirely filled in at this stage, however). The `Compose_Data_Request` function fills in certain default information into the header or header(s) of its input `Message`. The `Sequencing` function determines the sequence number used to uniquely identify the segment within a connection. The `Routing` function calculates the necessary information (such as the route identifier) that must be inserted into the header to reach the correct destination(s). After all this information is filled in, the PDU is suitable for transfer across the network. The `Retransmit` function ensures reliable delivery via a positive acknowledgment mechanism.

The next two functions shown in Figure 13 are used in both the sender and receiver protocol machines. The function `Connection_Establishment_Termination` performs connection set-up and tear-down. This function

```
(function "Segment"
  (mechanism "Segment"
    (input (Message sdu), (MTU_Type mtu_size))
    (output (Message sdu))
    (code "f_segment.o")
    (successors "Sequencing")))

(function "Compose_Data_Request"
  (mechanism "Compose_Data_Request"
    (input (Message sdu))
    (output (Message pdu))
    (code "f_compose_data_request.o")))

(function "Sequencing"
  (mechanism "Sequencing"
    (input (Message sdu))
    (output (Sequence_Number sn))
    (code "f_sequencing.o")
    (predecessors "Segment")))

(function "Routing"
  (mechanism "Transport_System_Routing"
    (input (Message sdu))
    (output (Routing_Result rr))
    (code "f_ts_routing.o")
    (predecessors "Receive_SDU")))

(timer "Retransmit"
  (mechanism "Timerbased_and_Cumulative_Retransmit"
    (input (Cumulative_Ack_List cal))
    (output (Message sdu))
    (predecessors "Connection_Establishment_Termination")
    (successors "Send_SDU")
    (code "f_timerbased_cumulative_retransmit.o")))

(function "Connection_Establishment_Termination"
  (mechanism "Explicit_Connection"
    (input (Message sdu)) (output (PDU_Header pdu))
    (code "f_explicit_connection.o))
  (mechanism "Implicit_Connection"
    (input (Message sdu)) (output (PDU_Header pdu))
    (code "f_implicit_connection.o")))

(function "Checksum"
  (mechanism "Checksum_Calculation"
    (input (Message pdu)) (output (PDU_Checksum cs))
    (code "f_checksum_calculation.o"))
  (mechanism "Checksum_Validation"
    (input (PDU_Checksum cs),
      (Message pdu)) (output (Boolean cbr))
    (code "f_checksum_validation.o")))

(function "Duplicate_Control"
  (mechanism "Duplicate_Control"
    (input (Message pdu))
    (output (Duplicate_Result dr))
    (code "f_duplicate_control.o")))

(function "Reassemble"
  (mechanism "Reassemble"
    (input (Message sdu))
    (output (Message sdu))
    (code "f_reassemble.o")))
```

Figure 13: Protocol Function Descriptors in the Protocol Resource Pool

descriptor contains two mechanisms, one used for explicit connections (such as a 2- or 3-way handshake) and one used for implicit connections (such as timer-based set-up and tear-down). The `Checksum` function also contains several mechanisms that may perform either checksum calculations or validations on a PDU.

The final two functions appear in the receiver-side of a protocol machine. The `Duplicate_Control` function detects and potentially discards replicated SDUs that arrive on the incoming data and control streams. The `Reassemble` function coalesces the individual pieces produced by the sender's `Segment` function to form the original PDU.

Figure 14 presents two more descriptor types: *anchor function* descriptors and *rendezvous* descriptors. The anchor function descriptors indicate the entry point(s) into and/or the exit point(s) out of protocol machine. For example, the `Upper_Interface_Sender` anchor function receives a

```
(anchor "Upper_Interface_Sender"
   (mechanism "Receive_SDU_Sender"
     (input (Message sdu))
     (output (Message sdu))
     (code "f_recv_sdu_sender.o"))
   (mechanism "Application_Interface"
     (input (Message sdu))
     (output (Message sdu))
     (code "f_application_interface.o")))

(anchor "Upper_Interface_Receiver"
   (mechanism "Send_SDU_Receiver"
     (input (Message sdu))
     (output (Message sdu))
     (code "f_send_sdu_receiver.o"))
   (mechanism "Application_Interface"
     (input (Message sdu))
     (output (Message sdu))
     (code "f_application_interface.o")))

(anchor "Lower_Interface_Sender"
   (mechanism "Send_SDU_Sender"
     (input (Message sdu))
     (output (Message sdu))
     (code "f_send_sdu_sender.o"))
   (mechanism "Network_Layer_Interface"
     (input (Message sdu))
     (output (Message sdu))
     (code "f_network_interface.o")))

(rendezvous "Validitiy"
   (mechanism "Audio_Validate"
      (input (Checksum_Result,
              Lifetime_Result, Duplicate_Result))
      (output (Boolean br))
      (code "f_audio_validate.o")
   (mechanism "Video_Validate"
      (input (Checksum_Result,
              Lifetime_Result, Duplicate_Result))
      (output (Boolean br))
      (code "f_audio_validate.o")
   (mechanism "Text_Validate"
      (input (Ack_Check_Result, Window_Check_Result,
              Path_Result, Checksum_Result))
      (output (Connection_State_Info,
               Exception_Handling_Info,
               Data_Reference, Option_Handling_Info,
               Round_Trip_Time, Congestion_Control_Info))
      (code "f_text_validate.o")
   (mechanism "Generic"
      (input (paramter_list)) (output (list_of_lists)))
```

Figure 14: Anchor and Control Function Descriptors in the Protocol Resource Pool

message from the application at the sender's end-system and logically and/or physically copies it into the internal memory space of the transport system. The rendezvous descriptor is a special-purpose control function that synchronizes concurrent processing within a protocol machine. A set of customized `Validate_Header` mechanism are defined in the protocol resource pool for existing protocol machine configurations. In order to configure new protocol machines automatically, a `Generic` mechanism is included. This mechanism is a "code template" that is parameterized by a list of input types and a list of output types. This template is used by the synthesis tools to automatically derive a suitable control mechanism.

Figure 15 depicts two other types of descriptors located in the protocol resource pool. These descriptors are composite types that contain portions of the protocol, control, and anchor function descriptors. For instance, a configuration descriptor characterizes the platform-independent configuration of a particular protocol machine and indicates whether it belongs to a specific application service class. The "code" for a composite configuration is stored as an internal intermediate representation in the protocol resource pool. Likewise, a protocol machine instantiation descriptor contains platform-

```
(configuration "Video_Stream"
  (class "real_time_unreliable")
  (code "video_stream.conf"))

(configuration "Text_Stream"
  (class "non_real_time_reliable")
  (code "text_stream.conf"))

(instantiation "Audio_Stream"
  (class "real_time_unreliable")
  (code "audio_stream.exe"))

(instantiation "Reliable_Message"
  (class "non_real_time_reliable")
  (code "reliable_message.exe"))
```

Figure 15: Protocol Machine Configuration and Instantiation Descriptors in the Protocol Resource Pool

dependent object-code and data that enables it to be executed on a particular target platform. In particular, the contents of an instantiation descriptor contain object-code attributes stored in the protocol, control, and anchor function descriptors.

5.4 Text-based Descriptions of Protocol Machines

This subsection illustrates several examples of a text-based language that utilizes various protocol resource descriptors to describe application-tailored protocol machine configurations. These configurations may either be produced manually or automatically (via protocol machine configuration tools [21]). In either case, the text-based protocol machine configuration is processed automatically by synthesis tools that transform it into a protocol machine instantiation. The text-based language illustrated in Figure 16 corresponds directly to the flowgraph-based language for the audio sender and the reliable message receiver shown in Figure 5. A configuration described via the text-based language consists of a collection of *clauses* that contain information (such as the successor and predecessor nodes) that are necessary to configure the protocol machine. The rendezvous and timer nodes are included to provide guidance for the instantiation process.

The anchor nodes in Figure 16 are represented by the `Upper_Interface_Receiver` clause, as well as the `Lower_Interface_Receiver` clause. Their predecessor and successor nodes are `NULL`, respectively, since they occur at the boundaries of the communication subsystem. Different mechanisms for the rendezvous node `Complete_Header` fill in the data and connection control packet header formation. Taken as a whole, these protocol mechanisms define the complete protocol machine for the audio sender.

Figure17 illustrates the text-based protocol machine configuration for the reliable message receiver shown in Figure 10. The anchor nodes for this protocol machine are also represented by the `Upper_Interface_Receiver` clause and the `Lower_Interface_Receiver` clause. The rendezvous node `Validate_Header` evaluates the results from its predecessor functions and selects the appropriate successor (either the `Acknowledgement` or `Reassemble`

```
(anchor "Upper_Interface_Sender"
   (mechanism "Receive_SDU_Sender"
     (predecessors NULL)
     (successors "Segment", "Synchronization")))

(anchor "Lower_Interface_Sender"
   (mechanism "Send_SDU_Sender"
     (predecessors "Complete_Header", "Checksum"")
     (successors NULL)))

(rendezvous "Complete_Header"
  (mechanism "Complete_Header_Audio_Data"
    (predecessors barrier
      ("Compose_Data_Request", "Sequencing",
       "Synchronization"))
    (successors "Send_SDU")))

(rendezvous "Complete_Header"
  (mechanism "Complete_Header_Audio_Connection"
    (predecessors barrier
       ("Connection_Establishment_Termination",
        "Routing"))
    (successors ("Checksum"))))

(function "Segment"
  (mechanism "Segment"
    (predecessors "Receive_SDU")
    (successors "Sequencing", "Compose_Data_Request")))

(function "Compose_Data_Request"
  (mechanism "Compose_Data_Request"
    (predecessors "Compose_Data_Request")
    (successors "Complete_Header)))

(function "Sequencing"
  (mechanism "Sequencing"
    (predecessors "Segment")
    (successors "Complete_Header)))

(function "Synchronization")
  (mechanism "Stream_Synchronization"
    (predecessors "Receive_SDU")
    (successors "Complete_Header")))

(function "Connection_Establishment_Termination"
  (mechanism "Explicit_Connection"
    (predecessors "Receive_SDU")
    (successsors "Complete_Header")))

(function "Routing"
  (mechanism "Transport_System_Routing"
    (predecessors "Receive_SDU")
    (successors "Complete_Header")))

(function "Checksum"
  (mechanism "Checksum_Calculation"
    (predecessors "Complete_Header")
    (successors "Send_SDU")))

(timer "Retransmit"
  (mechanism "Timerbased_and_Cumulative_Retransmit"
    (predecessors "Connection_Establishment_Termination")
    (successors "Send_SDU")))
```

Figure 16: Text-based Configuration of the Protocol Machine for Sending Audio

function). The protocol machine configuration may be submitted as input to the synthesis tools and/or stored in the protocol resource pool for subsequent instantiation.

6 Concluding Remarks

This paper examines a protocol machine configuration language and a protocol resource descriptor language that support a function-based approach for generating application-tailored protocol machines based upon a high-level description of protocol functionality. These protocol machines efficiently support a diverse collection of data streams required by multimedia applications such as the collaborative distance learning example presented in Section 3. In this example scenario, each different multimedia data stream is associated with an application-tailored protocol machine. The building-

```
(anchor "Upper_Interface_Receiver"
   (mechanism "Send_SDU_Receiver"
     (predecessors "Reassemble")
     (successors NULL)))

(anchor "Lower_Interface_Receiver"
   (mechanism "Receive_SDU_Receiver"
     (predecessors NULL)
     (successors "Duplicate_Control", "Ack_Check",
                 "Checksum_Calculation")))

(rendezvous "Validate_Header"
  (mechanism "Reliable_Msg_Validate"
     (predecessors barrier
       ("Duplicate_Control", "Ack_Check",
        "Checksum_Calculation"))
     (successors selector ("Reassemble",
                           "Acknowledgement"))))

(function "Checksum_Calculation"
  (mechanism "Checksumming"
    (predecessors "Receive_SDU")
    (successors "Validate_Header")))

(function "Duplicate_Control"
  (mechanism "Duplicate_Control"
    (predecessors "Receive_SDU")
    (successors "Validate_Header")))

(function "Acknowledge_Check"
  (mechanism "Acknowledge_Check"
    (predecessors "Receive_SDU")
    (successors "Validate_Header")))

(function "Reassemble"
  (mechanism "Reassemble"
    (predecessors "Validate_Header")
    (successors "Acknowledgement", "Send_SDU")))

(function "Acknowledgement"
  (mechanism "Negative_Selective_Ack"
    (predecessors "Validate_Header", "Reassemble")
    (successors "Retransmit")))
```

Figure 17: A Text-based Protocol Machine Configuration for the Reliable Message Receiver

block components for these configurations reside in a protocol resource pool, which contains a set of mechanisms used to generate a customized protocol machine. These machines are currently defined manually and are used as a basis for various synthesis and mapping tools. In addition, we are working on a set of configuration tools that will automate the process of producing the protocol machine configurations from high-level specifications of application qualitative and quantitative communication requirements [21].

The technique of using flowgraphs as a language for specifying function-based protocol machines is described in this paper using notions and definitions from the function-based communication model presented in [1]. In addition, the same approach is also applicable to the system described in [4].

References

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

[2] D. C. Schmidt, B. Stiller, T. Suda, A. Tantawy, and M. Zitterbart, "Language Support for Flexible, Application-Tailored Protocol Configuration," in *Proceedings of the* 18th *Conference on Local Computer Networks*, (Minneapolis, Minnesota), pp. 369–378, Sept. 1993.

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

[4] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment," *Journal of Concurrency: Practice and Experience*, vol. 5, pp. 269–286, June 1993.

[5] D. C. Schmidt, B. Stiller, T. Suda, and M. Zitterbart, "Tools for Generating Application-Tailored Multimedia Protocols on Heterogeneous Multi-Processor Platforms," in *Proceedings of the Second Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, (Williamsburg, Virgina), IEEE, September 1993.

[6] T. Braun and M. Zitterbart, "Parallel Transport System Design," in *Proceedings of the 4th IFIP Conference on High Performance Networking*, (Belgium), IFIP, 1993.

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

[8] D. L. Tennenhouse, "Layered Multiplexing Considered Harmful," in *Proceedings of the* 1st *International Workshop on High-Speed Networks*, May 1989.

[9] D. C. Feldmeier, "Multiplexing Issues in Communications System Design," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 209–219, ACM, Sept. 1990.

[10] D. Hehmann, M. Salmony, and H. Stuettgen, "Transport services for multimedia applications on broadband networks," *Computer Communications, Special Issue on Multi-Media Communications*, vol. 13, May 1990.

[11] W. P. Lidinsky, "Data Communication Needs," *IEEE Network Magazine*, pp. 28–33, March 1990.

[12] M. Anagnostuo, M. Theologou, K. Vlakos, D. Tournis, and E. Protonotarios, "Quality of service requirements in ATM-based B-ISDNs," *Computer Communications*, vol. 14, May 1991.

[13] O. Koufopavlou, A. N. Tantawy, and M. Zitterbart, "Analysis of TCP/IP for High Performance Parallel Implementations," in *17th Conference on Local Computer Networks*, (Minneapolis, Minnesota), Sept. 1992.

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

[15] 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.

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

[17] M. B. Abbott and L. L. Peterson, "A Language-Based Approach to Protocol Implementation," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Baltimore Maryland), ACM, 1992.

[18] R. Steinmetz, "Synchronization Properties in Multimedia Systems," *Journal on Selected Areas in Communications*, vol. 8, Apr. 1990.

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

[20] N. C. Hutchinson, S. Mishra, L. L. Peterson, and V. T. Thomas, "Tools for Implementing Network Protocols," *Software Practice and Experience*, vol. 19, pp. 895–916, September 1989.

[21] B. Stiller, "PROCOM: A Manager for an Efficient Transport System," in *Proceedings of the First IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Feb. 1992.