# Configuration Support for Flexible, Function-Based Communication Systems


Douglas C. Schmidt
Tatsuya Suda

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

Burkhard Stiller
Martina Zitterbart

Institute of Telematics
University of Karlsruhe
Karlsruhe, Germany

Ahmed Tantawy

T.J. Watson Research Center
IBM
Yorktown Heights, NY



An earlier version of this paper appeared in the proceedings of the IEEE $18^{th}$ Local Computer Networks conference in Minneapolis, Minnesota, September 1993.



## Abstract

*The next generation of communication subsystems must support diverse applications (such as interactive voice and video conferencing, supercomputer visualization, collaborative work, and remote process control) operating over high-performance local, metropolitan, and wide area networks (such as FDDI, SMDS, and B-ISDN). This paper describes a framework that contains a number of resources, languages, and tools for generating customized protocols that support diverse multimedia applications running on high-performance networks. This framework facilitates the configuration of application-tailored, function-based communication protocols that are automatically synthesized from high-level specifications. In addition, this framework also decouples the platform-independent aspects of protocol configuration from the platform-dependent aspects. This enables the generation of efficient protocols on a variety of hardware platforms that offer parallel processing based on shared memory and message passing architectures.*


## 1 Introduction

Communication systems must undergo significant changes to support the emerging diversity of application requirements and high-performance network characteristics efficiently and flexibly. This diversity is manifested by multimedia applications that contain multiple data streams (such as video, voice, and text) possessing different service characteristics. Likewise, the next generation of networks (such as DQDB, FDDI, and ATM-based B-ISDN) provide increased channel speeds, decreased bit error rates, and a wider range of services (such as asynchronous, synchronous and isochronous data delivery) than traditional networks.

Delivering these high-performance network capabilities to applications may require modifications to existing communication models, service interfaces, and protocol functionality. For example, layered communication models limit the potential for processing protocols on parallel platforms. They also include redundant functionality in multiple protocol layers [1]. Moreover, conventional transport service interfaces (such as those available in the OSI [2] and TCP/IP [3] protocol suites) do not enable applications to specify certain service requirements (such as isochronous data delivery or inter-stream synchronization). In addition, the next generation of protocols must efficiently support the increasing integration of traditional and emerging applications and network services. It appears to be infeasible, however, to develop a single "monolithic" protocol that integrates every different type of service. Such a protocol would likely become a highly complex edifice that is difficult to implement correctly and is unable to support all types of service efficiently.

A more flexible approach may be to develop "application-tailored" protocols that are customized for specific types of services such as transferring voice, video, text, and image data [4, 5]. Each of these protocols may be simpler and more efficient to design and implement than a single monolithic protocol [6]. However, an application-tailored approach may be impractical if substantial effort is required to configure and reconfigure each customized protocol implementation manually. This paper describes a framework that is being developed to help overcome this potential drawback. This framework provides specification notations and configuration languages that support the automated generation of customized protocols. These customized protocols are composed of reusable components known as *protocol function* [1]. Each protocol function performs a well-defined protocol processing task such as flow control, error control, acknowledgment, and connection establishment.

The framework presented in this paper facilitates the development of communication subsystems that adapt rapidly to changes in application requirements and advances in network technology. To achieve this, the languages and tools in this framework emphasize automation, reusability, and flexibility. Automated language support based on reusable building-block functions simplifies the development of cus-


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

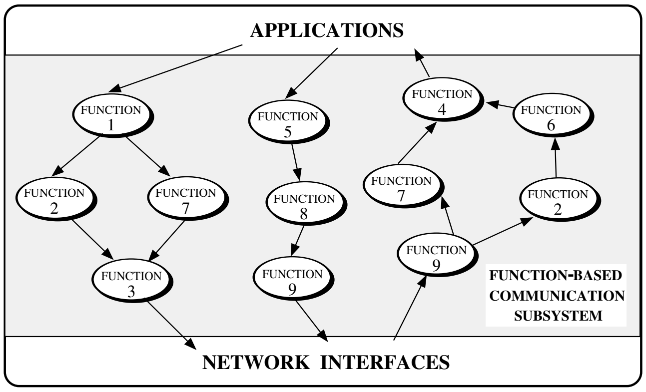


Figure 1: The Function-based Communication Model

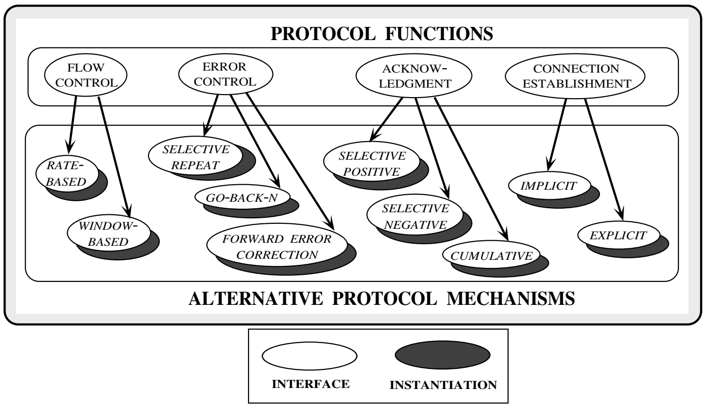


Figure 2: Protocol Functions and Protocol Mechanisms

tomized protocols. In contrast to manual programming techniques that use conventional low-level languages and operating system (OS) services directly, automation enables applications and developers to utilize the increased flexibility offered by an application-tailored protocol approach. Moreover, the framework presented here decouples the hardware and OS platform-independent aspects of protocol generation from the platform-dependent aspects. This decoupling helps facilitate the generation and execution of efficient protocols on a variety of hardware platforms (such as transputers and symmetric multi-processors) that offer several types of parallel processing architectures (such as message passing and shared memory).

A model describing a flexible function-based communication subsystem (F-CSS) is presented in [1]. A similar approach (ADAPTIVE) involving transport system support for multimedia applications is presented in [7]. Other related work addresses issues such as architectures that support function-based protocol decomposition [5, 8] and graph- and shape-based protocol configuration techniques [6, 9]. However, the related work does not address in detail the automated support necessary to generate application-tailored, function-based protocols that run on parallel platforms.

This paper is organized as follows. Section 2 outlines and motivates the function-based communication model and the framework that supports this model. Section 3 presents several languages that facilitate the description and configuration of conventional and customized protocols. Section 4 describes a higher-level notation for specifying application services via qualitative and quantitative requirements. Section 5 outlines the tasks performed by tools that support and automate the protocol generation process. Section 6 presents concluding remarks.

# 2 A Function-based Communication Subsystem

A communication model (such as the OSI reference model) describes a set of abstract entities and the relationships between these entities. This section outlines and defines the primary entities in a *function-based* communication model [1]. It also describes the major components in a framework that is being developed to implement the function-based communication model on several hardware and operating system platforms. The components in this framework include various resources (such as C++ component libraries and information bases), specification notations and configuration languages, and supporting tools. Although the framework described in this paper is oriented towards the function-based communication model, it is designed to support other types of traditional communication models as well [10].

## 2.1 The Function-based Model and Communication Architecture

Traditional communication models (such as the Internet and ISO OSI reference models) are characterized by hierarchical protocol *layers*. However, the efficiency, functionality, and flexibility of these layered communication models may be inadequate to support the increasing diversity of application requirements [11, 12]. These inadequacies are addressed by the function-based communication subsystem (F-CSS) described in [1]. F-CSS is based upon a function-based communication model characterized by a "de-layered" communication architecture. In this model, conventional coarse-grain hierarchical protocol layers are replaced by a communication subsystem that is decomposed into finer-grain *protocol functions* (illustrated in Figure 1).

Typical examples of protocol functions include flow control, error control, acknowledgment, and connection establishment. Each protocol function may be implemented via alternative *protocol mechanisms* (as shown in Figure 2). For instance, the flow control function may be implemented by either window-based and/or rate-based mechanisms. In essence, these mechanisms comprise the *instructions* of the virtual machine that constitutes the function-based communication model. Functions and mechanisms may be treated as "black-boxes" and accessed solely via their interfaces. Moreover, each mechanism may be instantiated using different implementation techniques (such as memory management schemes that minimize data copying, hardware support for protocol functions such as checksumming and demultiplexing, and parallel processing of protocol functions) available on various target platforms.

The primary motivations for utilizing a function-based approach are to enhance service flexibility and to improve performance in order to better satisfy application requirements. For example, 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 protocols that use protocol functions as their basic building-blocks. These protocols are specially-tailored to contain the minimal set of functions required to perform a particular service. In addition, functions form a convenient level of abstraction that is amenable to parallel execution on various multi-processor platforms. Performance measurements indicate that this function-based communication model is a promising approach for developing high-performance transport systems [13].

Protocol functions also 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 may be specifically-tailored to support an application *data stream*. A data stream represents a unidirectional flow of application data between one or more communicating entities. To support complex multimedia applications (such as teleconferencing), multiple data streams may be consolidated to form a *session*. A *session manager* coordinates the protocol machines of related data streams within each session. It generates and interprets session control information and performs various management tasks such as adding, modifying, or deleting data streams dynamically. Each data stream in a session is implemented by a protocol machine that is customized for a specific set of application requirements during a particular time period.

Figure 3 depicts the relationships between these various entities. 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 data stream is implemented by a different protocol machine, which is coordinated by a session manager in each session. The network interface is responsible for demultiplexing incoming application data onto a particular protocol machine that is associated with a unique combination of session and data stream. This de-layered approach enables the selected protocol machine to process incoming data without requiring additional demultiplexing operations, thereby reducing jitter [14] and increasing the potential for parallelism by minimizing synchronization overhead.

## 2.2 Configuration and Synthesis of Protocol Machines

The framework described in this paper provides language support for the automatic generation of function-based, application-tailored protocol machines. The languages in this framework access and manipulate information stored in a *protocol resource pool*. Among other things, this resource pool contains *descriptors* that characterize key syntactic and semantic attributes of alternative mechanisms that implement various protocol functions.

Applications and developers may compose protocol machines by indirectly or directly selecting particular functions and mechanisms from the resource pool. Protocol machines may be configured via several high-level and/or low-level notations and languages that are presented in Sections 3 and 4. These language descriptions are processed by tools (described in Section 5) that automatically generate customized protocol machines containing a particular set of protocol mechanisms.

The protocol generation and execution process may be characterized by several transformation phases (illustrated in Figure 4).[2] In the first phase, *configuration tools* generate *protocol machine configurations* based upon requests submitted via the service interface of the communication subsystem. A protocol machine configuration is an intermediate representation that indicates the particular mechanisms to apply in a particular sequence on incoming or outgoing application data in a stream. The configurations generated in the first transformation phase are designed to be independent of the underlying hardware and operating system platform. Therefore, they are not directly executable.

In the second phase, *synthesis tools* transform the configurations into *protocol machine instantiations*. An instantiation is an executable protocol machine that consists of resources (such as object code and related data) that may be optimized to run efficiently on a particular target execution platform (such as transputers [13] or shared memory multi-processors [7]). Instantiations are implemented via mechanisms selected from the protocol resource pool during the first transformation phase. Depending on the hardware platform, protocol mechanisms may be grouped into clusters and mapped onto one or more processing elements using the tools discussed in Section 5.

Configuration and synthesis tools generate protocol machines either *dynamically* (as an application is running) or *statically* (during an "off-line" protocol machine generation stage). Dynamic generation enables applications and/or the participating peer communication subsystems to modify certain protocol machine characteristics at run-time by using operating system facilities such as dynamic linking and object-oriented programming features such as dynamic binding. This enables applications to adaptively update protocol mechanisms and/or internal parameter values in a pre-defined

[2] These transformation phases are distinguished in this paper to clarify the presentation of the general architecture; a particular implementation may consolidate the phases to enhance performance.

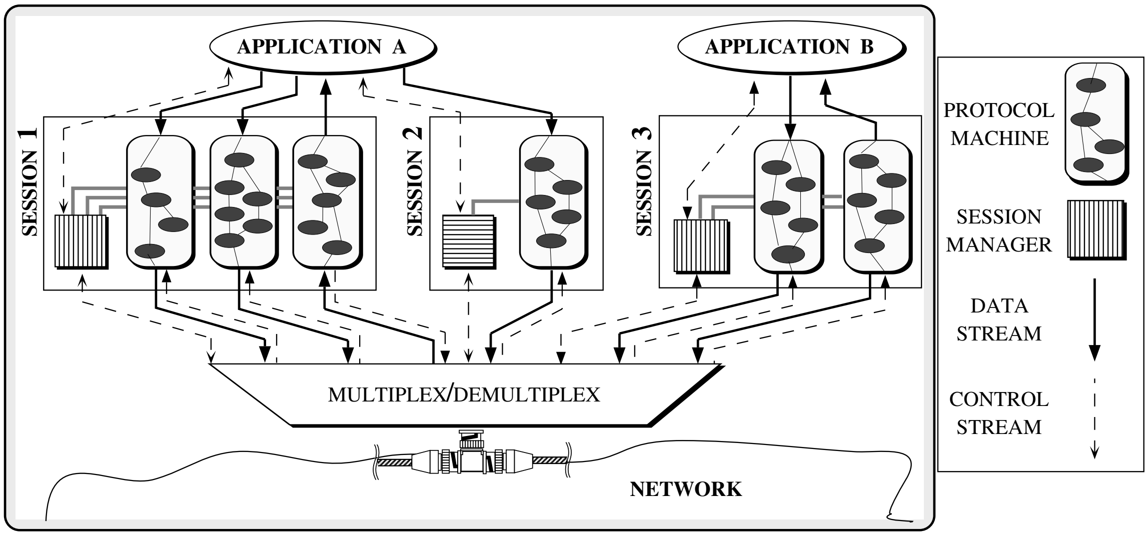

Figure 3: Relationship between Entities in the Function-based Communication Model

protocol machine instantiation. For example, an application may specify the maximum size of its data units when selecting a pre-defined instantiation. If a path discovery mechanism [15] determines that the maximum transmission unit of the underlying network supports this size without requiring fragmentation, the segmentation function may be removed from the instantiation at connection establishment time, before any processing of application data units occurs.

Static generation enables developers to "pre-define" protocol machines that implement various *application service classes*. These classes efficiently support different types of communication service such as unreliable real-time service (*e.g.,* isochronous traffic such as voice conversation and full-motion video), reliable real-time service (*e.g.,* robotics and process control), unreliable non-real-time service (*e.g.,* electronic mail) and reliable non-real-time service (*e.g.,* file transfer and remote login). Each protocol machine that implements one of these classes is configured to use the minimal set of protocol functionality required to perform that particular service.

The function-based framework leverages off the flexibility offered by the ability to invoke the configuration and synthesis tools dynamically and/or statically. For example, in addition to storing protocol function and mechanism building-blocks, the protocol resource pool also stores persistent pre-defined protocol machine configurations (which are not directly executable) and protocol machine instantiations (which are directly executable). Storing configurations is useful for developing a family of protocol machines that are designed to execute on a heterogeneous collection of target platforms. Rather than replicating the entire sequence of tasks required to produce a protocol machine configuration, the necessary configuration tasks are performed only once. The resulting configuration is a platform-independent intermediate representation that describes both the sequence and type of protocol mechanisms comprising the protocol machine. Subsequently, one or more target platform-specific instantiations may be generated by invoking synthesis tools to "fill-in" and interconnect the necessary platform-dependent mechanisms.

Pre-defined protocol machine instantiations may also be stored in the protocol resource pool. Storing instantiations in the resource pool reduces application start-up overhead at run-time since the time-consuming configuration and synthesis phases are avoided. Applications may directly or indirectly select one or more of these persistent instantiations when specifying their requirements via the service interface. The target platform is responsible for mapping the cluster components of a pre-defined instantiation onto the run-time system and executing them using one or more processing elements.

# 3 Languages for Configuring and Describing Protocol Machines

A protocol machine configuration contains a set of *descriptors* stored in the protocol resource pool, their predecessor and successor relations, and any synchronization information necessary to coordinate collaborating protocol functions. Each protocol resource descriptor indicates certain attributes of the function such as its name, the specific mechanism(s) that implement the function, the input and output parameters of each mechanism, and any dependencies the mechanism(s) may have on system resources (such as the amount of available memory).

The mechanisms in the protocol resource pool form an *instruction set* that may be "programmed" via several notations and languages that exist at different levels of abstraction. The highest level notation specifies application requirements via the service interface of the communication subsystem. The next level provides flowgraph-based and text-based lan-

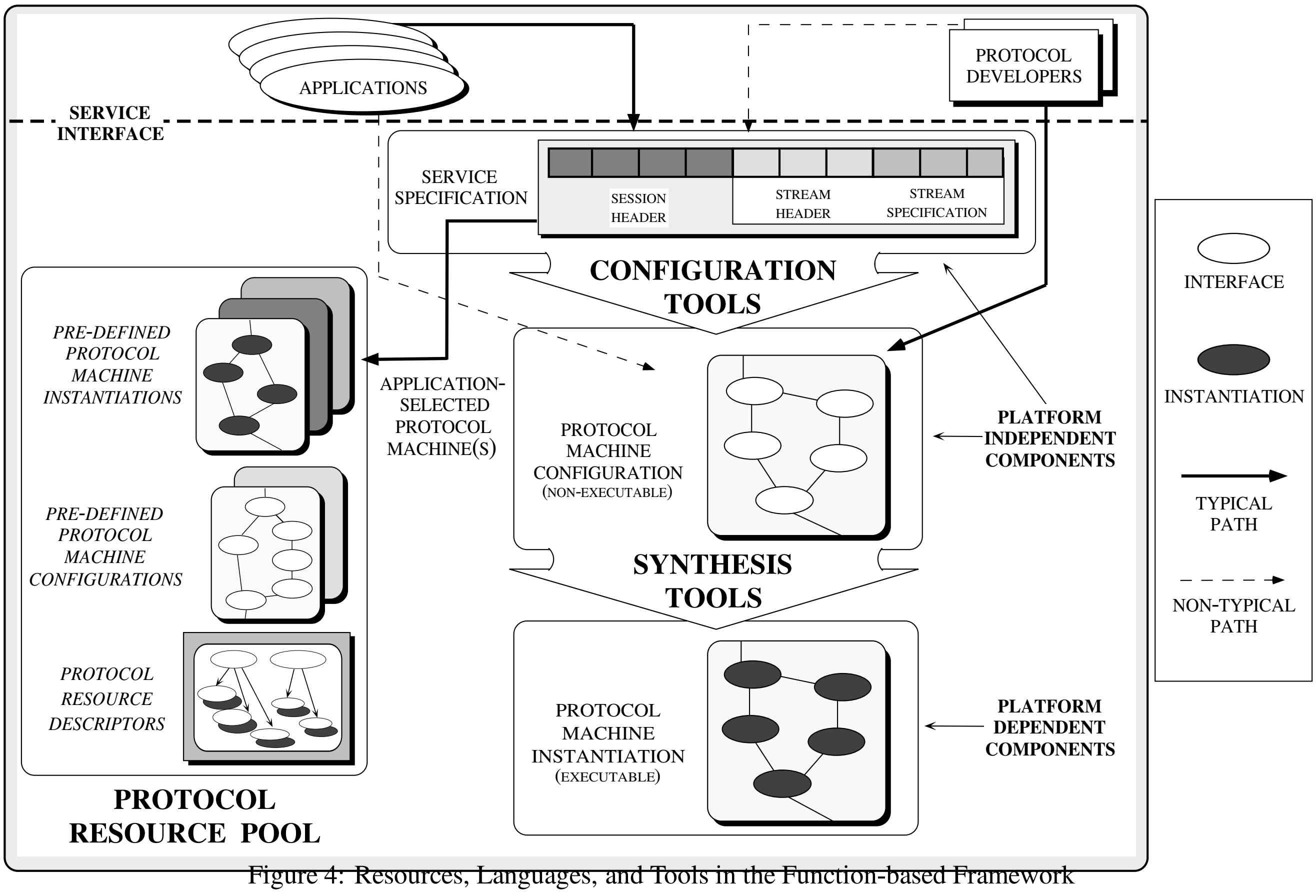


Figure 4: Resources, Languages, and Tools in the Function-based Framework

guages used to configure protocol machines that are generated from functions and mechanisms residing in the protocol resource pool. This pool is populated using a *protocol resource descriptor language* that characterizes the essential syntactic and semantic aspects of each protocol function. At the lowest level, protocol mechanisms are implemented on a particular target platform using conventional programming languages (such as C and C++). Figure 5 illustrates the levels of abstraction between the various languages in the framework.

The framework provides a range of languages that enable applications and developers to generate protocols using the appropriate level of abstraction, rather than requiring them to program directly in low-level conventional languages such as C or C++. Typically, applications use the service specification notation to indicate their qualitative and quantitative service requirements. This high-level notation reduces the amount of details that applications must consider and specify at run-time. If a pre-defined instantiation exists that meets these requirements, it is directly selected. Otherwise, configuration and synthesis tools are invoked dynamically to generate a suitable protocol machine. Protocol developers, on the other hand, may prefer to describe protocol functionality using the lower-level configuration languages. These languages provide developers with fine-grained control over the selection of particular protocol functions and/or mechanisms. Customized protocol machines generated by developers may be stored persistently as pre-defined instantiations in the protocol resource pool. Applications may subsequently invoke these instantiations at run-time to fulfill their service requirements.

Section 3.1 illustrates the main features of the flowgraph-based protocol machine configuration language that specifies the segment reception portion of the TCP protocol. Section 3.2 discusses the language used to populate the protocol resource pool with protocol resource descriptors.

## 3.1 Protocol Machine Flowgraphs

The framework described in this paper provides both a flowgraph-based and a text-based language for describing protocol machine configurations. The flowgraph-based representation uses different types of nodes to visually express properties, attributes, and predecessor/successor relationships between functions that constitute a protocol machine configuration. Constructs in the flowgraph-based representation translate essentially "one-to-one" onto the text-based language. Protocol developers may prefer the flowgraph-based format since various graphical-editing tools simplify the configuration process by handling bookkeeping details (such as keeping track of the interconnections among predecessors and successors). Conversely, configuration tools [16] may use the text-based format directly since it is more

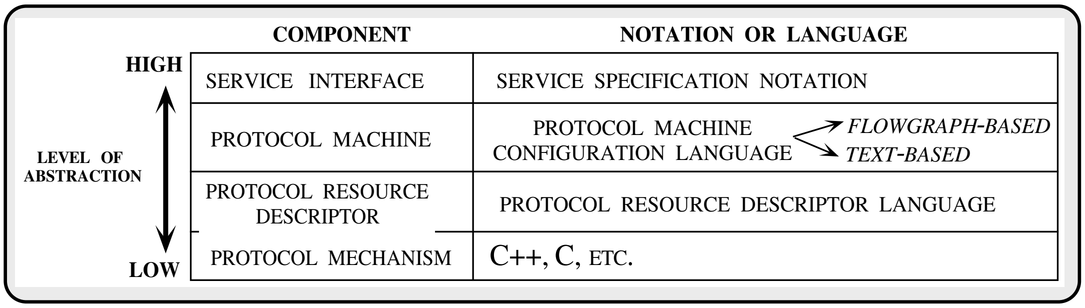


Figure 5: Levels of Abstraction for Notations and Languages

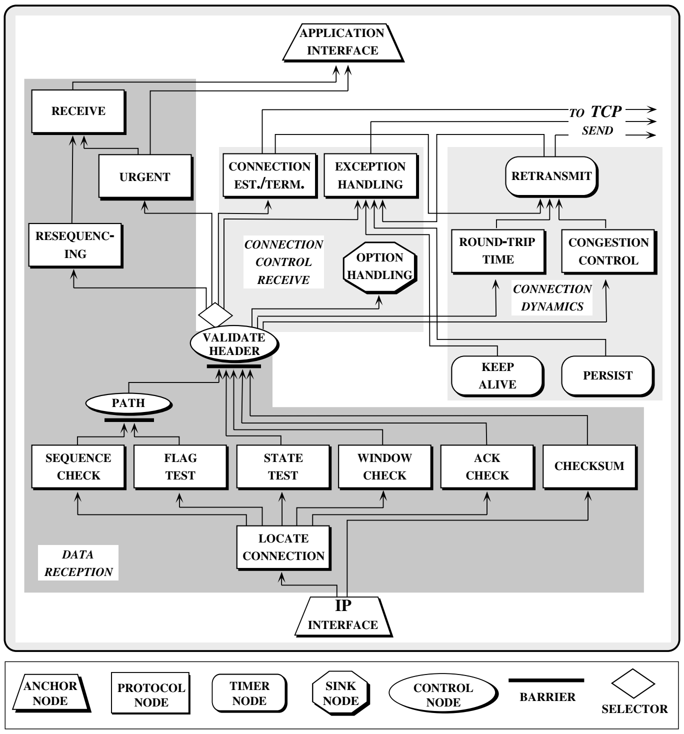


Figure 6: A Flowgraph for the TCP Receiver Protocol Machine

efficient to manipulate programmatically via automated processing tools.

The flowgraph-based configuration depicted in Figure 6 formalizes the notation used to depict function-based decompositions of TCP and IP in related publications [17, 18]. These decompositions illustrate the potential parallelism in the TCP and IP protocols. TCP is used as the example in this paper since it is a standard protocol with well-known behavior. Moreover, by configuring TCP function relationships via flowgraph constructs, the main features of the language are demonstrated. However, the flowgraph language is not limited to configuring only existing protocols. In fact, it is primarily intended to configure application-tailored protocol machines.

As with other types of flowgraphs, the constructs in a protocol machine flowgraph are *nodes* and *edges*. Each node in Figure 6 represents a single protocol function such as resequencing, checksum calculation, congestion control, option handling, and connection establishment and termination (Section 3.1.1 describes the different types of nodes in detail). The edges characterize the flow of control between the functions. The following discussion uses the flowgraph-based language to illustrate a protocol machine configuration for a portion of the TCP protocol that handles the reception of segments passed up the protocol stack from IP.

The TCP protocol machine for receiving segments is illustrated as three shaded components in Figure 6 that handle *data reception*, *connection control reception*, and various *connection dynamics* involving timer-related processing [18]. The description here focuses primarily on the data reception component (which consists of the nodes in the darkly shaded portion of the Figure). This component processes correctly received TCP segments containing regular and urgent application data segments.

To expedite protocol processing, the flowgraph is designed to exploit potential parallelism among the TCP checksum calculation function and the header decomposition functions that access the segment associated with a connection. For example, functions such as SEQUENCE CHECK, WINDOW CHECK, and ACK CHECK may concurrently determine whether the sequence and acknowledgement numbers are within their respective window ranges. Likewise, other header decomposition functions such as FLAG TEST may also inspect the incoming segment concurrently to determine whether any exception flags (such as FIN, SYN, RST, or URG) are enabled. As shown in Figure 6, these functions subsequently rendezvous at the VALIDATE HEADER node, which decides where control is transferred next. The decision depends on both the state of the connection and the flags in the segment. Assuming that no errors occur, the connection dynamics and OPTION HANDLING functions may be invoked simultaneously to update round-trip time estimates and process any options in the TCP segment header. Likewise, the appropriate reception function (either regular data, urgent data, or connection control reception) specified by the segment flags is also invoked. Urgent data may be made available to the application immediately, whereas regular data segments must first be processed by the RESEQUENCING function. In either event, the RECEIVE function must update the connection's flow control window before passing the data up to the application interface.

### 3.1.1 Graphical Notation

The edges and nodes in the protocol machine flowgraph language express key properties and attributes of protocol mechanisms. These properties and attributes form a type system that provides valuable information to configuration and synthesis tools. For instance, edges depict potential mechanism interactions and indicate whether a protocol function may transfer control to only one or to many successors simultaneously. For example, the edges emanating from the LOCATE CONNECTION node in Figure 6 indicate that the header decomposition functions may be invoked concurrently once the appropriate connection identifier is located.

Node types indicate whether a function is invoked directly (via the regular sequence of processing steps) or indirectly (via timer control). During the configuration phase, this type information simplifies the syntactic and semantic analysis required to correctly and automatically configure the necessary functions and mechanisms. For instance, data dependencies between the input and output parameters of a mechanism are used to constrain the potential predecessor and successor relations of a particular protocol node. The relations between functions in the example presented here are fixed in advance by the TCP protocol specification. In contrast, the ordering relations between nodes in application-tailored protocol machines are often more flexible [16].

During the synthesis phase, type information characterizing data dependencies may be analyzed further to determine opportunities for parallel execution of certain protocol mechanisms. In addition, when mapping a protocol machine onto a particular target platform, type information helps guide the selection of efficient platform-dependent techniques for interconnecting protocol mechanisms together at run-time on one or more processing elements.

Several types of nodes are distinguished according to the general class of functions they perform. The function inside a node is treated as a "black-box," *i.e.,* it is represented only via its interface rather than its instantiation. The following paragraphs describe the various types and subtypes of nodes that may appear in a protocol machine flowgraph:

• **Anchor Nodes:** *Anchor nodes* represent functions that occur at the entry and exit points into and out of a protocol machine. Anchor nodes are generally located at the service access points within and around the communication subsystem. In the de-layered function-based communication model, anchor nodes typically interact only with entities at the boundaries of the communication subsystem. For example, they may transfer service data units to and from network interfaces and user applications. In the TCP example, they occur at service access points between IP and TCP, as well as between TCP and the application interface. Depending on various factors (such as the network/host interface, the protection-domain scheme, and the degree of multi-programming and multiplexing supported by the end-system), anchor nodes may perform functions that copy service data units from the network and place them into subsystem memory buffers, initiate cross-domain mode-switches, utilize various synchronization primitives to protect shared system resources, and demultiplex PDUs onto the appropriate protocol machine.

• **Protocol nodes:** *Protocol nodes* represent protocol functions (such as segmentation/reassembly, lifetime control, checksum calculation, retransmission, routing, and flow control) that access and/or modify *information units*[3] that flow through a protocol machine. Protocol nodes may be further classified into several subtypes:

- *Timer Nodes* – Timer nodes represent certain timer-driven protocol functions that are invoked "asynchronously" with respect to the regular flow of control in a protocol machine. For example, timers are used to invoke the TCP retransmission function (as well as the acknowledgement function in certain protocols [19]). A protocol machine flowgraph may not explicitly indicate a direct transfer of control to timer nodes (*i.e.,* there may be no incoming edges into the node). Therefore, the use of a distinct node subtype provides information necessary to properly analyze the flowgraph and transform the protocol machine configuration into a correct implementation.
- *Sink Nodes* – Sink nodes represent protocol functions that do not propagate information units outside a protocol machine. For example, functions such as TCP option handling (or error handling functions for unreliable protocols such as UDP) simply update certain internal context information in the protocol machine and return. Sink nodes are characterized graphically by the absence of any outgoing edges.

These subtypes are used internally by configuration and synthesis tools to facilitate syntactic and semantic analysis.

In general, the commonalities between these different subtypes may be expressed via object-oriented techniques such as inheritance (which facilitates software reuse) [9] and dynamic binding (which decouples mechanism interfaces from mechanism algorithms and defers certain implementation decisions until run-time). These techniques facilitate a modular, extensible, and efficient object-oriented software architecture for the configuration and synthesis tools, as well as for the run-time environment [10].

• **Control Nodes:** *Control nodes* represent functions that coordinate and synchronize protocol and anchor nodes in the protocol machine. *Rendezvous nodes* [18] are a subtype of control nodes that represent special *pseudo*-functions that

[3] Information units consist of application data and protocol control packets, along with units that report the status of local and remote sessions and deliver stream management requests to a session manager [1].

synchronize concurrent processing in the flowgraph. Typically, rendezvous nodes do not perform actual protocol functions. Instead, they validate or evaluate the results of concurrently executed predecessor nodes in order to determine the next action(s) to perform.

In addition to different types and subtypes, protocol machine configuration flowgraphs may also contain several auxiliary symbols. These symbols characterize the input and output semantics associated with the transfer of control between predecessors and successors of one or more nodes. For example, a **barrier** dictates that all incoming control from predecessor nodes must synchronize before any further processing occurs. The rendezvous node labeled VALIDATE HEADER in Figure 6 illustrates the use of a barrier that synchronizes the header decomposition and checksum functions. If a barrier is not present, control may be transferred into a node from multiple predecessor mechanisms without requiring synchronization. The EXCEPTION HANDLING node in the TCP example illustrates this case. Likewise, a node with more than one outgoing edge may represent a decision point in the control flow of a protocol machine. If a **selector** is not present, then all the successor nodes may be invoked simultaneously. Otherwise, control may be transferred to only one of the multiple successors. For instance, the VALIDATE HEADER node contains a selector that chooses between the RESEQUENCING, URGENT, and CONNECTION ESTABLISHMENT AND TERMINATION functions.

The flowgraph *statically* depicts all successors that a node may *potentially* transfer control to at run-time. During run-time, the protocol function at a node will *dynamically* select the particular successor nodes(s) that will perform subsequent processing. The edge(s) selected at run-time depend on factors such as the success or failure of a processing operation or characteristics of an information unit (such as its size or the value of a field in its control header). Judiciously placing these barriers and selectors at synchronization points and decision points in the flowgraph allows the configuration processing tools generate more time and space efficient protocol machine implementations. For example, generating code for a node with only a single successor may run more efficiently and require less storage since control transfer decisions need not be performed at run-time.

#### 3.1.2 Relation to Existing Graphical Notations for Specifying Protocols

Flowgraphs are commonly used as architectural description and translation techniques in many computing domains such as VLSI synthesis and compiler optimization. In the communication subsystem domain, the *x*-kernel project utilizes a graph-based notation called a *protocol graph* to describe and implement "highly-layered" protocols [6]. The two basic types of nodes in a protocol graph correspond to *virtual*- and *micro*-protocols. A virtual-protocol uses information such as the protocol address in a message header or the message size to determine which subsequent micro-protocol to invoke. The selected micro-protocol then processes the associated message. Edges in a protocol graph imply the transfer of a message between interconnected micro-protocols.

Morpheus is a special-purpose programming language that utilizes the OS facilities of the *x*-kernel as a run-time environment [9]. Protocol developers utilize object-oriented techniques like inheritance to compose protocols constructed from various "shapes." The three classes of shapes in Morpheus (*multiplexers*, *routers*, and *workers*) are distinguished by the type of addressing information they expect. Different shapes embody certain semantic information that is specific to the communication protocol domain. Therefore, a compiler for the Morpheus language may perform certain optimizations (such as reducing the overhead of passing PDUs via multiple protocol layers that are interconnected at run-time) that are not detectable via conventional programming language compilers. In addition, the compiler generates code that reuses pre-existing *x*-kernel protocol support library components (such as hash tables and message managers), thereby simplifying the overall protocol development effort.

There are several differences between the type systems offered by the *x*-kernel protocol graphs and Morpheus shapes and the type system offered by the protocol machine flowgraphs described in Section 3.1. Most notably, the flowgraph-based configuration language enables the specification of data dependency and synchronization information that may be used to detect and exploit parallelism in protocol machines. Neither the *x*-kernel nor Morpheus have any explicit provisions for expressing parallelism in their protocol graphs. Specification languages such as Petri-nets [20] also provide graphical notations that specify the potential for concurrent processing. However, the effort required to extend Petri-nets to express the semantics necessary to automate the generation of application-tailored, function-based protocol machines is significant enough to warrant the development of the languages described in this paper.

Other specification languages such as Estelle, Lotus, SDL, and PASS provide notations that are used to specify protocol behavior at the finite-state machine level [21]. However, the flowgraph language described in this paper provides a representation that is particularly intended to describe function-based protocol machines. For instance, protocol machine flowgraphs directly characterize the functionality and the ordered predecessor/successor relationships provided by the protocol mechanisms residing in the protocol resource pool. This correspondence helps to simplify the development of configuration and synthesis tools.

### 3.2 Protocol Resource Descriptor Language

Configuration and synthesis tools utilize syntactic and semantic information associated with *protocol resource descriptors* that reside in the protocol resource pool. Descriptors in this pool possess attributes that guide both the transformation of service specifications into protocol machine configurations, as well as the subsequent transformation of configurations

```
<protocol-resource-descriptor> ::=
  <protocol-function-descriptor> |
  <anchor-function-descriptor> |
  <control-function-descriptor>  |
  <configuration-descriptor> |
  <instantiation-descriptor>

<protocol-function-descriptor>
  '('FUNCTION <function-name>
    { '(' MECHANISM <mechanism-name>
          '('  INPUT  <input-parameter-list> ')'
          '('  OUTPUT <output-parameter-list> ')'
          '('  CODE <code-for-mechanism> ')'
        [ '('  PREDECESSORS <predecessor-node-list> ')'
          '('  SUCCESSORS   <successor-node-list>   ')'
          '('  CONSTRAINTS   <condition_list>        ')' ]
       ')' }
   ')'

<instantiation-descriptor>
  '(' INSTANTIATION <instantiation-name>
        [ '(' CLASS <class-name>    ')' ]
          '(' CODE  <code-path-name> ')'
   ')'
```

Figure 7: Extended-BNF Format for a Protocol Resource Descriptor

into executable instantiations. To facilitate interaction with the resource pool, the framework provides a language that describes certain attributes associated with protocol functions and the mechanisms that implement these functions.

Figure 7 illustrates a portion of the general syntax for several descriptors via an extended-Backus/Naur Format (EBNF) notation. Each resource descriptor contains a name (*e.g.,* `Flow_Control`) that uniquely identifies it in the resource pool. One or more mechanisms (e.g., `Sliding_Window` or `Rate_Control`) are associated with each function (the EBNF notation for "one or more instances of" are indicated by the left and right curly-brackets "{" and "}" in the figure). Each mechanism contains both mandatory and optional attributes. Mandatory attributes must be included with every mechanism description. The mandatory attributes of protocol function descriptors include the object code (*e.g.,* `sliding_window.o` or `rate_control.o`), as well as the input and output parameters of a mechanism. For example, the input parameters for the `Sliding_Window` mechanism might include the segment identifier, sequence number, and credit update.

The predecessor, successor, and constraint attributes are optional (indicated by the EBNF "zero or one" left and right square-bracket notation "[" and "]" in the Figure), and may be necessary only in certain circumstances. For example, developers who populate the resource pool with protocol resource descriptors may need to specify additional semantic constraints. These constraints limit the conditions under which a particular mechanism may be used, and may also restrict the predecessor and successor relations. If no constraints or predecessor/successor restrictions apply to a function or mechanism, these optional attributes may be entirely omitted from the descriptor.

Semantic constraints provide guidance to configuration and synthesis tools that enables them to generate correct and efficient protocol machines automatically. Constraints are

```
(function "Locate_Connection"
  (mechanism "TCP_Locate_Connection"
    (input (Source_Port sp), (Destination_Port dp))
    (output (Connection_ID cid))
    (code "f_tcp_locate_connection.o")
    (predecessors "IP_Interface")))

(function "Checksum_Calculation"
  (mechanism "TCP_Checksum"
    (input (PDU_Checksum cs), (PDU pdu))
    (output (Boolean cbr))
    (code "f_tcp_checksum.o")))

(function "Urgent"
  (mechanism "TCP_Urgent"
    (input (Urgent_Flag uf), (Urgent_Pointer up))
    (output (SDU sdu))
    (code "f_tcp_urgent.o")))

(function "Sequence_Check"
  (mechanism "TCP_Sequence_Check"
    (input (Sequence_Number sn)) (output (Boolean sbr))
    (code "f_tcp_seq_check.o")
    (predecessors "Locate_Connection"))
  (mechanism "Gen_Sequence_Check"
    (input (Sequence_Number sn)) (output (Boolean sbr))
    (code "f_gen_seq_check.o")))

(function "Flow_Control"
  (mechanism "Sliding_Window"
    (input (Segment_ID sid), (Sequence_Number sn),
           (Credit_Update cu))
    (output (Segment_ID sid), (Lower_Edge le),
            (Actual_Credit ac))
    (code "f_sliding_window.o"))
  (mechanism "Rate_Control"
    (input (Segment_ID sid), (Rate_Update rd),
           (Burst_Update bd))
    (output (Segment_ID sid), (Rate r), (Burst b),
            (Rate_Timer rt))
    (code "f_rate_control.o")))
```

Figure 8: Protocol Resource Descriptors in the Protocol Resource Pool

necessary since there are certain situations where a mechanism's interface does not convey enough information to adequately restrict the appropriate predecessor and successor relations. For example, the `TCP_Sequence_Check` mechanism requires the connection ID to be located before the sequence number if checked. However, since there is no direct parameter association between these two functions, additional information must be provided explicitly by the resource descriptor to enable the language configuration and synthesis processing tools to produce correct instantiations. Likewise, if a mechanism's input parameter is a composite type (such as an "information unit"), it may be impossible for tools to automatically deduce the dependencies or constraints this mechanism may have on other mechanisms.

A protocol resource descriptor may explicitly enumerate some or all of its potential predecessors and/or successors. This may be necessary to correctly generate protocol machines for standard protocols that possess tightly-coupled functionality. However, in an application-tailored model (such as that one provided by the function-based communication subsystem), excessive use of explicit enumeration may over-constrain the possible relationships between functions. Since this decreases the potential for parallelism and limits the reusability of the protocol mechanisms, developers are encouraged to minimize the constraints and dependencies placed on descriptors they insert into the resource pool.

Developers and tools may use the protocol resource descriptor language to insert, modify, access, and extract certain attributes of descriptors. For example, developers use

```
(anchor "IP_Interface"
  (mechanism "IP_SAP")
  (successors "Locate_Connection", "Checksum"))

(function "Locate_Connection"
  (mechanism "TCP_Locate_Connection")
  (predecessors "IP_Interface")
  (successors "Sequence_Check", "Flag_Test", "State_Test",
              "Window_Check", "Ack_Check"))

(function "Sequence_Check"
  (mechanism "TCP_Sequence_Check")
  (predecessors "Locate_Connection")
  (successors "Path"))

(function "Checksum_Calculation"
  (mechanism "TCP_Checksum")
  (predecessors "IP_Interface")
  (successors "Validate_Header"))

(function "Urgent"
  (mechanism "TCP_Urgent")
  (predecessors "Validate_Header")
  (successors "Application_Interface"))

(sink "Option_Handling"
  (mechanism "TCP_Options")
  (predecessors "Validate_Header"))

(rendezvous "Validate_Header"
  (mechanism "TCP_Validate")
  (predecessors barrier ("Path", "State_Test", "Window_Check",
                          "Ack_Check", "Checksum"))
  (successors selector ("Connection_ET", "Exception_Handler",
                         "Urgent", "Resequencing"),
              "RTT", "Congestion_Control", "Option_Handling"))
```

Figure 9: A Portion of a Text-based Protocol Machine Configuration for TCP

the descriptor language to insert protocol functions and their constituent mechanisms and attributes into the resource pool. Subsequently, certain attributes may require modification to reflect changes to descriptors such as replacing the object code of a mechanism with a more efficient implementation. Likewise, high-level configuration tools [16] access the semantic attributes of descriptors when determining how to correctly transform a service specification into a protocol machine configuration. In addition, synthesis tools extract the object code corresponding to each platform-dependent mechanism selected to generate a protocol machine instantiation. After all the selected mechanisms are interconnected, the target platform's run-time environment maps them onto one or more processing elements.

Figure 8 presents an example list of resource descriptors that may be used to populate a protocol resource pool with certain protocol functions and their associated mechanisms and attributes. The mechanisms in the first three descriptors are associated with the TCP example from Section 3.1. The first mechanism in the `Sequence_Check` function is also specific to the TCP example. However, the second mechanism in this function, as well as both mechanisms in the `Flow_Control` descriptor, is intended for general use with application-tailored protocol machine configurations. The protocol resource descriptor language enables mechanism dependency information to be indicated implicitly via the input and output parameters that form each mechanism's interface. For example, the `Sequence_Number` input parameter for the `Sliding_Window` mechanism implementing the `Flow_Control` protocol function implies a dependency on the `Sequence_Check` function. Various tools in the framework construct dependency graphs that process this information during the configuration and synthesis phases [1].

Figure 9 illustrates a portion of the TCP protocol shown in Figure 6 that is described via a text-based representation of the protocol machine configuration language. A configuration described via this text-based language consists of a collection of *clauses* that each contain information (such as the successor and predecessor nodes) that is necessary to configure the protocol machine. Note how constructs from the flowgraph-based configuration language (such as barriers and selectors) map concisely onto the text-based configuration language. Moreover, the syntax of the text-based configuration language is very similar to the resource descriptor language, with the exception of platform-dependent attributes (such as the object code) which are omitted in the configuration, as well as additional control nodes and synchronization information (which may be added to the configuration by developers and/or the configuration tools). The close correspondence between a text-based configuration language and a related object-descriptor language is also reflected in other interface-description languages such as CORBA [22]. CORBA uses a syntax that closely resembles C++ to describe service interfaces for distributed applications.

# 4 Service Specification Notation

In addition to the lower-level protocol machine configuration and resource descriptor languages, a higher-level *service specification notation* is also available. This notation provides a concise and convenient service interface that exports certain properties of the communication subsystem to applications and protocol developers. The service interface of the F-CSS system offers a wider range of services to applications than is generally available via conventional service interfaces (such as TLI [23] and sockets [24]). In particular, conventional interfaces typically offer only a limited choice of services (such as connection-oriented and connectionless services), and permit little or no capability for customizing the underlying protocol machines.

The F-CSS service interface passes specifications of qualitative and quantitative application service requirements to configuration and synthesis tools that have access to the protocol resources available at communication end-systems. Different actions may be taken depending on the type of specification provided by an application or developer. For example, an application may request a particular pre-defined *application service class*. If the existing protocol machines do not meet application requirements, however, tools may automatically generate a customized protocol machine from mechanisms residing in the protocol resource pool.

As shown in Figure 10, a service primitive consists of a fixed-length *session header* that specifies the attributes a session must support. Each data stream may be described via two additional fixed-length sets of fields known as the *stream definition*. This definition contains a *stream header* that includes the application service class, the mandatory bit, and

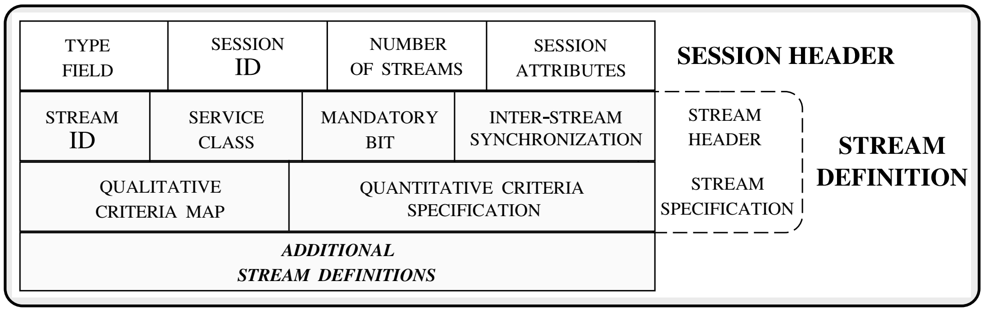


Figure 10: Structure of a Service Primitive

the inter-stream-synchronization specification fields. Likewise, it also contains a *stream specification* that indicates the list of qualitative parameters (such as ordered delivery, guaranteed group delivery, or syntax selection) and quantitative parameters (such as maximum throughput, minimum delay, or maximum error loss rate). There may be multiple data streams specified for each session.

To enable convenient specification of service primitives, many fields in the service interface may be expressed in a binary "enable/disable" manner. For example, qualitative parameters may be selected by enabling various fields in the application service primitive. At configuration time, configuration tools use a simple pattern matching algorithm to identify the particular protocol functionality requested by a service user. The details of the application service interface are described further in [1].

# 5 Tools for Generating and Executing Protocol Machines

This section describes several classes of tools that 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. Figure 11 illustrates the complete set of relationships between the resources and tools involved in this transformation process. The tool components access and manipulate the descriptors in the protocol resource pool to transform platform-independent descriptions 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. The synthesis and mapping tools perform the platform-dependent transformations, whereas the configuration tools are intended to be platform-independent. Each class of tools consults information residing in the protocol resource pool on the local end-system. The remainder of this section outlines the functionality offered by the tools in each class and describes the types of information they utilize from the descriptors residing in the protocol resource pool. In addition to the tools described below, various platform-specific operating system utilities (such as compilers and assemblers for conventional programming languages, as well as linkers and loaders) are also necessary to generate and execute application-tailored protocol machines.

## 5.1 Configuration Tools

Configuration tools transform high-level application service specification requests (submitted via the service interface shown at the top of Figure 11) into protocol machine configurations that are described via the flowgraph-based or text-based protocol machine configuration languages defined in Section 3. Configuration tools perform operations involving the *selection* and *ordering* of protocol resources [16]. The selection process determines which functions and mechanisms in the protocol resource pool are necessary to fulfill a particular application service request. The ordering process determines the predecessors and successors of each function and mechanism. Selection and ordering decisions are based on semantic information associated with protocol resource descriptors (such as input and output parameters and constraints), as well as domain-specific knowledge of communication protocols possessed by configuration tools (such as the minimal set of protocol functions required to satisfy a particular class of application service requests).

## 5.2 Synthesis Tools

Synthesis tools transform protocol machine configurations produced by the configuration tools into protocol machine instantiations. Synthesis tools perform operations involving the *composition* and *static interconnection* of protocol resources to form one or more *clusters*. A complete protocol machine instantiation consists of a set of interconnected clusters (depicted in Figure 12 (1)). Each cluster contains a set of platform-specific object-code extracted from the function descriptors in the protocol resource pool that were selected earlier by the configuration tools. The composition process uses the protocol machine configuration to guide the formation of one or more clusters. The static interconnection process determines efficient mechanisms for transferring control between functions within a cluster, as well as between interconnected clusters at run-time.[4]

[4]Clusters may also be interconnected dynamically (cf. Section 5.3).

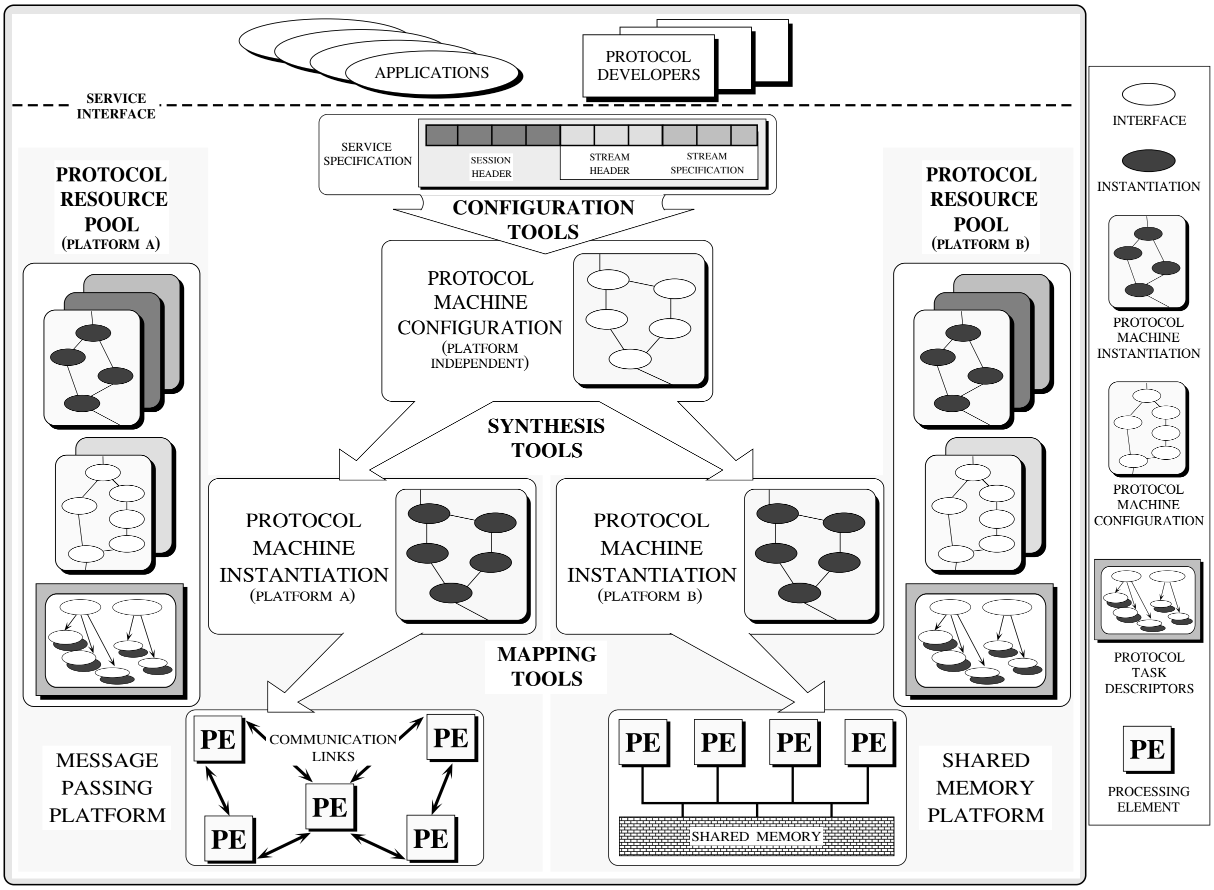


Figure 11: Resources and Tools used during Protocol Generation and Execution

Several alternative mechanisms may be used to transfer control between and within functions and clusters. For example, within a cluster, control is typically transferred between functions as a consequence of the hardware updating a program counter to reference the next executable protocol function or instruction. Mechanisms for transferring control between clusters, in contrast, depend on the underlying process architecture, operating system, and hardware platform. For instance, interprocess communication (IPC) mechanisms may be necessary to transfer control between functions in different clusters that are executing on separate processing elements in a non-shared memory platform. Conversely, if several clusters are executing concurrently on separate threads in a shared address space, control may be transferred between functions by simply traversing a pointer link to the next cluster. Depending on the underlying process architecture, synchronization primitives may be necessary to protect resources shared between concurrently executing threads of control. Pointer links between clusters may be established either *statically* (by the synthesis tools during protocol machine instantiation) or *dynamically* (by the mapping tools at run-time, as described in Section 5.3 below).

Protocol functions constitute the primary units of execution in a protocol machine, whereas clusters (which may contain one or more protocol functions) are the primary units of mapping and interconnection onto a particular target platform. There are several motivations for clustering certain protocol functions together in a protocol machine instantiation. In general, clusters decouple the *processing* of protocol functions from the *interconnections* that link the functions together. This decoupling facilitates *reuse*, *automation*, and *flexibility*. For example, fine-grain reuse of protocol functions is encouraged by developing the functions independently from how, when, or in what order they are eventually composed. Likewise, the mapping tools described below may be used to determine and perform the appropriate type of interconnections between clusters without requiring explicit intervention from developers or applications. In addition, flexibility is enhanced by deferring certain interconnection decisions until late in the protocol design process, potentially during installation or run-time.

By deferring these configuration-related decisions, communication subsystem may select more efficient mechanisms for interconnecting functions and clusters. Selecting an efficient interconnection mechanism depends both on *static* factors (such as bus bandwidth or whether the underlying target platform hardware architecture supports IPC via message passing and/or shared memory), as well as *dynamic* factors (such as the current system load on a particular processing element).

Determining the policies and mechanisms for clustering protocol resources is a research challenge. Potential crite-

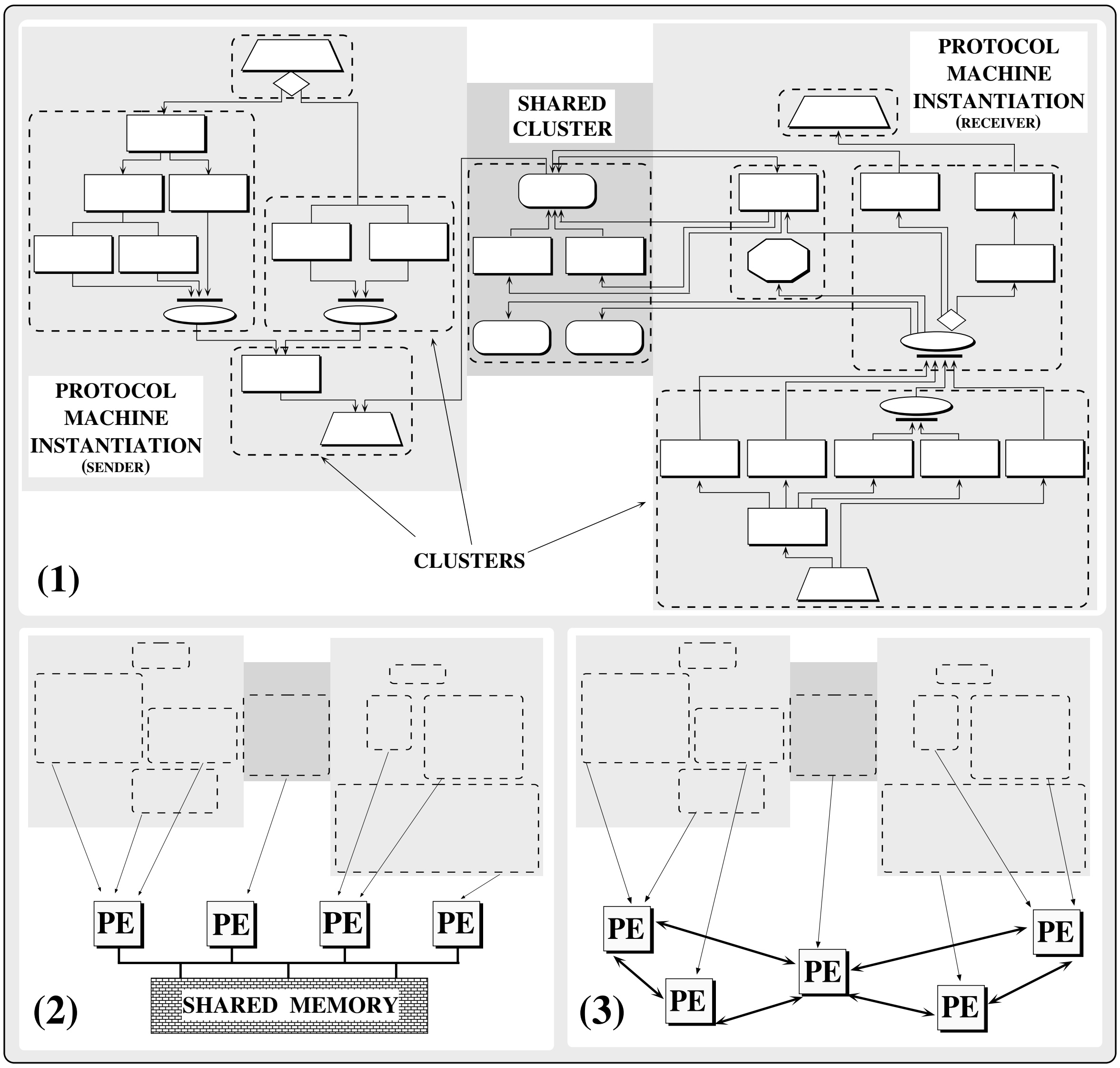


Figure 12: Clustering and Mapping Protocol Resources onto Multi-Processing Platforms

ria for partitioning protocol machines into clusters include (1) grouping resources to facilitate stage balancing within a multi-processor function "pipeline," (2) localizing functions that reference common data in order to minimize memory contention, exploit processor cache affinity, and/or reduce paging, (3) coalescing certain functions together to conveniently add or remove clusters from protocol machines at run-time, and (4) enabling more thorough static analysis of concurrent activity between and within clusters [25]. Future work will utilize the synthesis and mapping tool mechanisms to determine which cluster partition policies are suitable under which circumstances.

## 5.3 Mapping Tools

Mapping tools transfer the clusters that comprise a protocol machine instantiation into the run-time system of a particular target platform. The primary operations performed by mapping tools involve *local system resource allocation*, *dynamic interconnection*, and *cluster placement*. For instance, when an application activates one or more protocol machines, the mapping tools load the object-code associated with the clusters of the selected protocol machine instantiation(s) into the target platform's run-time system. This task involves allocating resources (such as memory and processing elements) and performing any necessary dynamic interconnections between clusters in the protocol machine instantiation.

Permitting the dynamic interconnection of clusters enables the reconfiguration of certain mechanisms in a pre-defined protocol machine instantiation. The capability to reconfigure protocol machines at run-time enables the communication subsystem to dynamically adapt to changes in network and application characteristics. For example, adaptive protocol error handling mechanisms may achieve a lower average transmission delay compared with non-adaptive approaches [26].

Mapping tools are also responsible for placing clusters onto the processing elements (PEs) available on the hardware platform. Determining the placement of clusters onto

PEs is crucial for achieving high levels of application and transport system performance. In order to determine a suitable mapping onto the PEs, the synthesis and mapping tools must cooperate to determine which clusters to associate with which PEs. In general, the synthesis tools described in Section 5.2 identify protocol function clusters and the mapping tools subsequently decide where to place these clusters within the underlying PE topology. The cluster placement process may be modeled via a graph description of the end-system's PE topology. This "placement graph" is labeled with the current load statistics (calculated as the quotient of the PE processing time versus the sum of idle and processing time) at the *nodes* (*i.e.,* PEs), and the current communication behavior of inter-processor connections at the *edges* (*i.e.,* communication links between PEs).

Clusters must be mapped onto the placement graph without exceeding limits on PE-load or memory resources. Therefore, the mapping tools must account for the anticipated PE utilization and the related communication and synchronization behavior. In addition, the mapping tools require detailed knowledge of certain static and dynamic hardware features (such as the number of available PEs, the mechanisms used to interconnect and communicate between the PEs, and maximum and current load capacity of the PEs). After the mapping operations are performed, the target platform's operating system is responsible for managing the scheduling and context switching of the protocol machine clusters during run-time.

Although the target platforms supported by this framework differ in terms of operating system and hardware aspects (such as the number of available PEs, the interprocess communication and memory architecture, and the network interface devices), many of the same resources, languages, tools, and underlying architectural principles may be applied on the different platforms. Using the graphical notation defined in the Section 3.1.1, Figure 12 illustrates the mapping of an identical set of clusters (Figure 12 (1)) onto a shared memory multi-processor platform (Figure 12 (2)), as well as onto a message-passing multi-processor system (Figure 12 (3)). We are currently experimenting with protocol implementations to characterize the advantages and disadvantages of each platform.

# 6 Concluding Remarks

The framework described in this paper provides a set of resources, languages, and tools that support the automated generation of protocol machines composed of reusable, function-based protocol mechanisms. Due to the emphasis on flexibility, extensibility, and reusability, this framework is particularly well-suited to support the increasing diversity of services required by multimedia applications. In particular, it is possible to reduce protocol development effort and facilitate rapid prototyping and experimentation with communication subsystems on a variety of hardware and operating system platforms. Moreover, performance measurements indicate that the function-based communication model is a promising architecture for developing high-performance communication subsystems. 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 [7].

# 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] OSI Standard, *OSI Transport Protocol Specification*, 1986.

[3] J. Postel, "Transmission Control Protocol," *Network Information Center RFC 793*, pp. 1–85, Sept. 1981.

[4] J. Sterbenz and G. Parulkar, "AXON: Application-Oriented Lightweight Transport Protocol Design," in *International Conference on Computers and Communications*, (New Delhi, India), Nov. 1990.

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

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

[7] 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.

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

[9] 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.

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

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

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

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

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

[15] J. Mogul and S. Deering, "Path MTU Discovery," *Network Information Center RFC 1191*, pp. 1–19, Apr. 1990.

[16] 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.

[17] A. N. Tantawy and M. Zitterbart, "Multiprocessing in High Performance IP Routers," in *3rd International IFIP WG 6.1/6.4 Workshop on Protocols for High-Speed Networks*, (Stockholm), IFIP, May 1992.

[18] 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.

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

[20] J. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice Hall, 1981.

[21] G. v. Bochmann, “Usage of Protocol Development Tools: the Results of a Survey,” in *Protocol Specification, Testing, and Verification, VII*, 1987.

[22] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1.2 ed., 1993.

[23] Sun Microsystems, *Network Interfaces Programmer's Guide*, Chapter 6 (TLI Interface) ed., 1992.

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

[25] D. L. Levine and R. N. Taylor, “Metric-driven reengineering for static concurrency analysis,” in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '93)*, (Cambridge, Mass), ACM press, June 28-30 1993.

[26] H. K. Huang, T. Suda, G. Takeuchi, and Y. Ogawa, “Protocol Reconfiguration: a Study of Error Handling Mechanisms,” in *Proceedings of the* 2$^{nd}$ *International Conference on Computer Communication Networks*, (San Diego, California), ISCA, June 1993.