Tools for Generating Application-Tailored Multimedia Protocols on Heterogeneous Multi-Processor Platforms

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

An earlier version of this paper appeared in the proceedings of the Second Higher Performance Communication Subsystems Workshop in Williamsburg, Virgina, September 1993.

Abstract

This paper describes an integrated set of tools for generating application-tailored communication 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 multi-processor end-system architectures. An overview of the steps used to generate customized protocol machines is presented and a set of criteria for characterizing tool behavior are defined. Three distinct classes of tools are examined: (1) configuration tools transform high-level application specifications into platform-independent protocol machine configurations by selecting and ordering the requisite protocol functions, (2) synthesis tools compose and statically interconnect platform-dependent object-code and data to form executable protocol machine instantiations, and (3) mapping tools place these executable instantiations onto processing elements in the run-time environment of shared-memory or message-passing target platforms.

1 Introduction

Developing high-performance communication subsystems is becoming increasingly necessary to efficiently and flexibly support the diversity of emerging multimedia applications (such as scientific visualization and computer-supported collaborative work projects [1]) running on high-performance local and wide-area networks (such as FDDI and ATM-based B-ISDN). However, conventional communication models are limited by static layering architectures that introduce redundant functionality and limit the potential for parallel processing [2]. Likewise, many conventional protocols do not offer a sufficiently diverse range of functionality (such as multicasting [3], inter-stream synchronization [4], or adaptive error handling [5, 6]). Moreover, it is difficult to modify the functionality of existing protocol implementations since they are typically developed in a monolithic and inflexible manner [7].

One approach for overcoming the limitations of conventional communication models and protocols is to develop application-tailored protocols that execute efficiently on a variety of hardware and operating system platforms [8]. This paper describes an integrated framework of tool components that are being developed to facilitate the generation of application-tailored protocols. The components in this framework automate many steps required to generate customized protocols and to execute these protocols in parallel on heterogeneous platforms (such as message passing transputers [9] and shared memory multi-processors [10]).

The work described in this paper is based on the principles of a function-based communication model that decomposes protocols into de-layered protocol function and mechanism components [2]. The primary objectives of this function-based model are to (1) enhance service flexibility and (2) increase the opportunities for processing protocol functions in parallel [11]. For instance, applications may specify their qualitative and quantitative requirements via a flexible communication subsystem service interface that guides the selection and/or generation of application-tailored protocols [8]. Likewise, protocol functions form a convenient and flexible level of abstraction that is amenable to parallel execution on various multi-processor platforms. Performance measurements indicate that the function-based communication model is a promising approach for developing high-performance transport systems [9].

The paper is organized as follows: Section 2 summarizes the configuration and description languages and protocol resources that are used by the tool components examined in this paper; Section 3 discusses three classes of tools for mapping platform-independent descriptions of customized protocols onto particular multi-processor platforms; and Section 4 presents concluding remarks.

[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.

```
<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 1: EBNF Format for Protocol Resource Descriptors

2 Configuration Language and Protocol Resource Components

The application-tailored protocols discussed throughout this paper are composed of reusable protocol function building-blocks such as connection establishment, segmentation, retransmission, segmentation, reassembly, sequencing, and checksumming. These functions serve as building-block *resources* that may be combined in a de-layered manner to generate efficient application-tailored *protocol machines* [2]. A protocol machine consists of a set of mechanisms (drawn from a *protocol resource pool*) that constitute the minimal set of functionality required to perform a particular application service (such as transferring voice, video, text, or image). Certain characteristics of individual protocol mechanisms and composite protocol machines may be specified via flowgraph-based and/or text-based protocol configuration and description languages that manipulate descriptors in the resource pool. For example, the *protocol resource description language* (defined in [8]) characterizes functions in terms of attributes such as mechanisms, parameters, semantic constraints, and object-code. An extended Backus/Naur (EBNF) format that portrays the general structure of protocol resource descriptors is shown in Figure 1.

The protocol resource pool also contains protocol machine *configurations* and *instantiations*. Configurations are non-executable, platform-independent descriptions that contain a set of protocol resource descriptors, their predecessor and successor relations, and an indication of the synchronization information necessary to coordinate interactions between the protocol resources at run-time. Instantiations, on the other hand, are executable protocol machines consisting of platform-dependent resources (such as protocol mechanism object-code and related data) that may be optimized to run efficiently on a particular target platform. Storing these components in the resource pool helps reduce application start-up overhead at run-time since some or all of the time-consuming configuration and synthesis phases may be elided.

Figure 2 illustrates how various *tools* (such as configuration and synthesis tools) and *resources* (such as protocol resource descriptors and pre-defined protocol machine configurations and instantiations) are described and manipulated by various specification notations and configuration languages. This figure also depicts the manner in which tools may be activated to select or generate protocol machines. For example, applications may describe their qualitative and quantitative application service requirements using a *service specification notation*, which is then submitted via the service interface of the communication subsystem. If a pre-defined protocol machine instantiation exists that meets these requirements it is directly selected. Otherwise, the configuration and synthesis tools (described in Section 3) may be invoked dynamically to generate a suitable protocol machine from resource descriptors residing in the protocol resource pool.

Figure 3 illustrates a text-based configuration depicting the sender-side of an audio protocol machine that utilizes descriptors in the protocol resource pool. In this example, Anchor Clauses (such as `Upper_Layer_Interface` and `Lower_Layer_Interface`) characterize the entry or exit access points of the protocol machine configuration, where data and control will be passed into or out of the transport system. Protocol processing is indicated via Function Clauses such as `Segment`, `Sequencing`, and `Routing`. Rendezvous Clauses (such as `Complete_Header`) are used to synchronize concurrent protocol processing. Moreover, functions that operate asynchronously under timer control (such as `Retransmit`) are specified as Timer Clauses.

Commonalities between these different types of clauses may be expressed via object-oriented techniques such as inheritance [12] (which facilitates software reuse) and dynamic binding (which both decouples mechanism interfaces from the mechanism algorithms and defers certain implementation decisions until run-time). These techniques facilitate a modular, extensible, and efficient object-oriented software architecture [13] for the configuration, synthesis, and mapping tools described in the following section.

3 Tool Classes

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 2 illustrates the relationships between the resources and tools involved in this transformation process.[2] 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.

[2]The transformation phases presented in this section are intended to clarify the essential characteristics of the model. However, a given implementation may consolidate one or more of these phases to enhance performance.

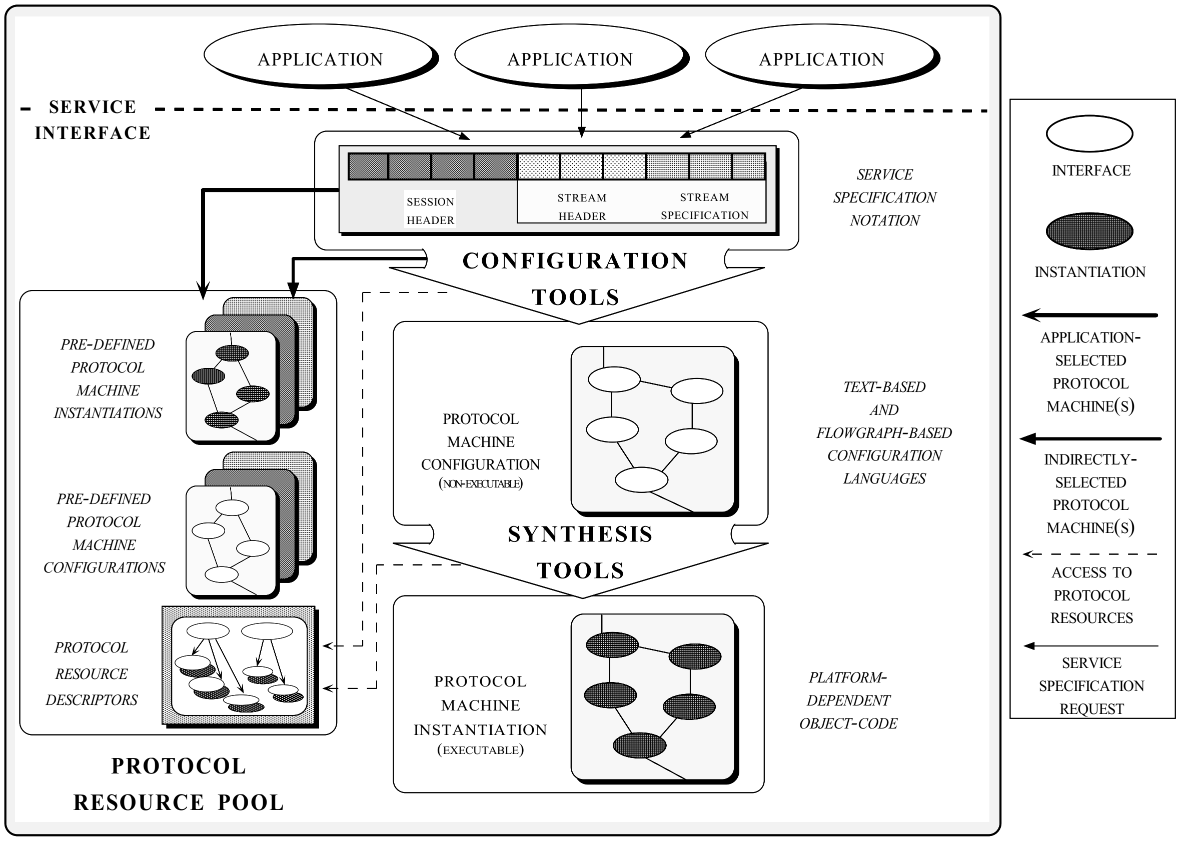
APPLICATION
APPLICATION
APPLICATION
SERVICE INTERFACE
SESSION HEADER
STREAM HEADER
STREAM SPECIFICATION
SERVICE SPECIFICATION NOTATION
CONFIGURATION TOOLS
PRE-DEFINED PROTOCOL MACHINE INSTANTIATIONS
PRE-DEFINED PROTOCOL MACHINE CONFIGURATIONS
PROTOCOL RESOURCE DESCRIPTORS
PROTOCOL RESOURCE POOL
PROTOCOL MACHINE CONFIGURATION (NON-EXECUTABLE)
TEXT-BASED AND FLOWGRAPH-BASED CONFIGURATION LANGUAGES
SYNTHESIS TOOLS
PROTOCOL MACHINE INSTANTIATION (EXECUTABLE)
PLATFORM-DEPENDENT OBJECT-CODE
INTERFACE
INSTANTIATION
APPLICATION-SELECTED PROTOCOL MACHINE(S)
INDIRECTLY-SELECTED PROTOCOL MACHINE(S)
ACCESS TO PROTOCOL RESOURCES
SERVICE SPECIFICATION REQUEST

Figure 2: Selection and Generation of Protocol Machines

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.

3.1 Configuration Tools

Configuration tools transform high-level application service specification requests (submitted via the service interface shown at the top of Figure 2) into protocol machine configurations that are described via the flowgraph-based or text-based protocol machine configuration languages defined in [8]. Configuration tools perform operations involving the *selection* and *ordering* of protocol resources [14]. 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).

3.2 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 4 (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

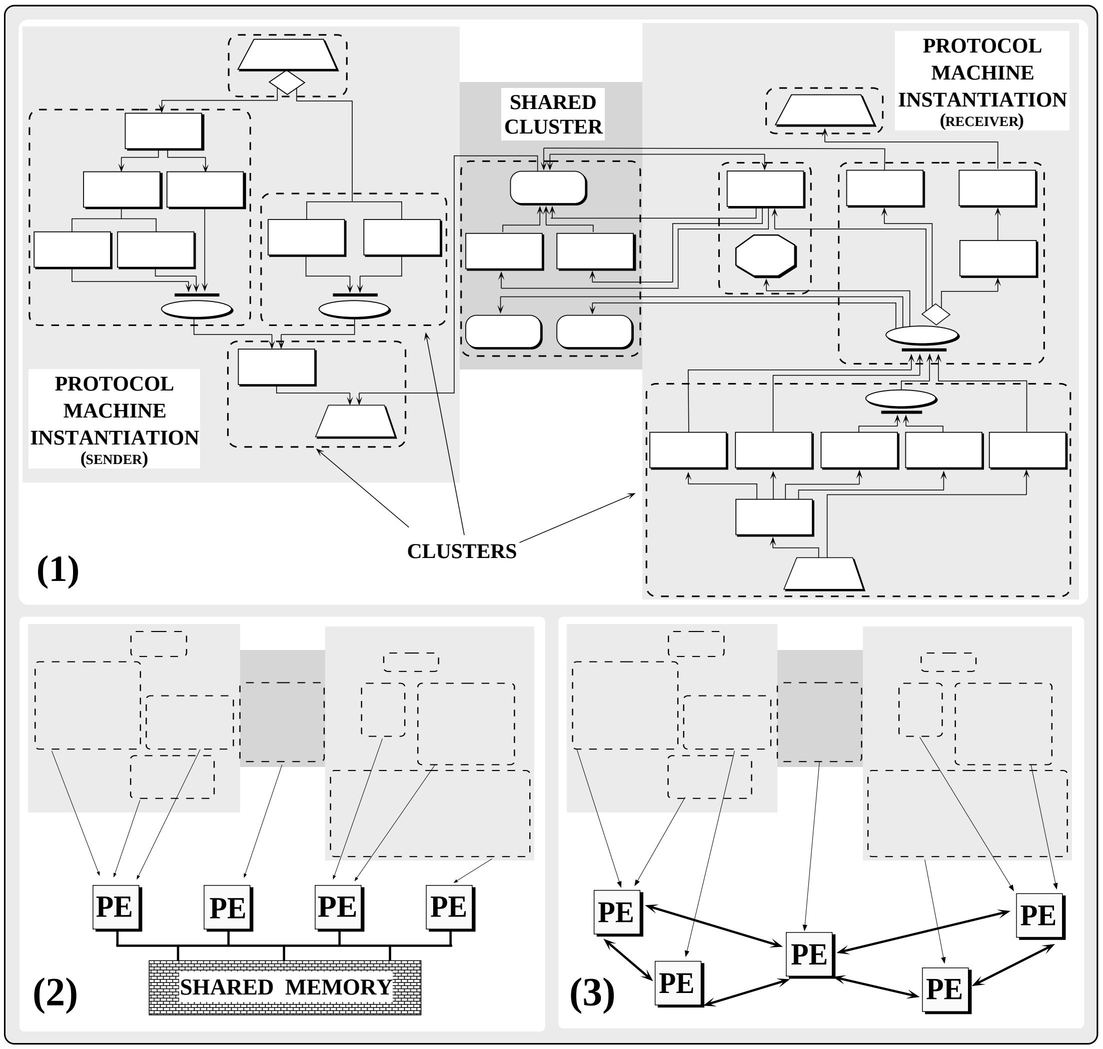
PROTOCOL MACHINE INSTANTIATION (RECEIVER)
SHARED CLUSTER
PROTOCOL MACHINE INSTANTIATION (SENDER)
CLUSTERS
(1)
PE
SHARED MEMORY
(2)
(3)

Figure 4: Clustering and Mapping Protocol Resources onto Multi-Processing Platforms

interconnected clusters at run-time.[3]

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, on the other hand, 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 3.3 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

[3]Clusters may also be interconnected dynamically (cf. Section 3.3).

```
(anchor "Upper_Interface_Sender"
   (mechanism "Receive_SDU_Sender"
     (predecessor-node NULL)
     (successor-node "Segment", "Synchronization")))

(anchor "Lower_Interface_Sender"
   (mechanism "Send_SDU_Sender"
     (predecessor-node "Complete_Header", "Checksum"")
     (successor-node 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", "Segment")
    (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_Calculation"
  (mechanism "32_Bit_Checksum"
    (predecessors "Complete_Header")
    (successors "Send_SDU")))

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

Figure 3: Text-based Configuration of the Protocol Machine for Sending Audio

during installation or run-time. By deferring these 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 facing both developers and tools. Potential criteria 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 [15]. Future work will utilize the synthesis and mapping tool mechanisms to determine which cluster partition policies are suitable under which circumstances.

3.3 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. For example, an application may specify the maximum size of its data units when selecting a pre-defined instantiation. If a path discovery mechanism [16] 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. Likewise, 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 [6].

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 3.2 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 notation defined in [8], Figure 4 illustrates the mapping of an identical set of clusters (Figure 4 (1)) onto a shared memory multi-processor platform (Figure 4 (2)), as well as onto a message-passing multi-processor system (Figure 4 (3)). We are currently experimenting with protocol implementations to characterize the advantages and disadvantages of each platform.

4 Concluding Remarks

This paper describes an integrated framework being developed to generate application-tailored protocol machines. The protocol resources, configuration languages, and supporting tools in this framework facilitate the transformation of platform-independent protocol machine configurations into platform-dependent instantiations. The synthesis and mapping tools discussed throughout the paper utilize platform-specific information that describes the static and dynamic characteristics of the underlying OS and hardware. The tools use this information to interconnect clusters of protocol functions and to place them onto the run-time system of shared memory and message passing multi-processors.

References

[1] C. A. Ellis, S. J. Gibbs, and G. L. Rein, "Groupware: Some Issues and Experiences," *Communications ACM*, vol. 34, pp. 38–58, Jan. 1991.

[2] 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.

[3] S. E. Deering and D. R. Cheriton, "Multicast routing in datagram internetworks and extended LANs," *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 85–110, May 1990.

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

[5] T. L. Porta and M. Schwartz, "Performance Analysis of MSP: a Feature-Rich High-Speed Transport Protocol," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (San Francisco, California), IEEE, 1993.

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

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

[8] 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.

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

[10] 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.

[11] 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.

[12] M. B. Abbott and L. L. Peterson, "A language-based approach to protocol implementation," *IEEE Journal of Transactions on Networking*, vol. 1, Feb. 1993.

[13] 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.

[14] 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.

[15] 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.

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