Beyond Layering: Modularity Considerations for Protocol Architectures

Kenneth L. Calvert
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30032-0280
calvert@cc.gatech.edu

Abstract

This paper considers ways to achieve modularity in protocol architectures without layering. Modularity promotes interoperability by making it easier to port protocol implementations; layering, on the other hand, has been identified as a performance impediment because it enforces sequential processing of messages. Focusing on end-to-end protocols, we propose mechanisms—metaheaders and generic interfaces—that support modular parallel or integrated protocol implementations and encourage architecture-independent specifications. An example illustrates interoperability between two systems that support the same protocols, one using traditional layering and the other supporting parallel protocol composition.

1 Introduction

As transmission speeds approach a gigabit per second and applications demand ever-greater communication capacity, it is sometimes argued that current protocol architectures are becoming a performance bottleneck. In particular, the basic paradigm of *layering* has been identified as an impediment to performance [3, 5, 12]. Accordingly, some researchers have proposed integration of all end-to-end protocol functions into a single layer that spans the gap between the application and the network [3, 7]. The idea is that *integrated* implementations can achieve better performance without the sequential processing constraints enforced by layering.

Interestingly, some recent work [11] proposes a protocol architecture emphasizing *increased* layering. Such an architecture enjoys the traditional benefits of modularity, including flexibility and ease of development of new protocols and services. However, even with various performance optimizations it is subject to the fundamental constraint of layering, namely that messages must be processed sequentially by the protocol layers.

This paper is an attempt to reconcile these apparently divergent points of view. It is motivated by the observation that proposals for new protocol architectures typically assume that all communicating systems will implement the new architecture; interoperability with other architectures is seldom a design consideration. This is a shame because of the following Law of Protocol Architectures: *There exist installations that need to communicate but do not support the same protocol architecture.* In other words, there will always be systems that cannot inter-operate without some "out-of-channel" effort—say, porting a protocol implementation from one to the other. It is therefore desirable for the next generation of protocol architectures to minimize the cost of *gaining* interoperability for systems that do not initially have it. But unless layering is replaced by some other structuring paradigm, the overall cost of achieving interoperability will *increase* because of the difficulty of porting complex, highly-tuned protocol implementations. We therefore consider ways to encourage modular implementations of end-to-end protocols while avoiding the known performance pitfalls of layering.

The rest of the paper is organized as follows. Section 2 reviews different architectural characteristics that have been proposed for performance and modularity. Section 3 presents a general model of protocol function and shows how layered encapsulation enforces sequential processing; the role of *protocol glue*—i.e., functions related to protocol composition that do not directly affect interoperability—is highlighted. The use of *metaheaders* to support parallel header processing is proposed in Section 4. Section 5 discusses the use of *generic interfaces* in eliminating architectural

0-8186-3670-X/93 $03.00 © 1993 IEEE

assumptions from protocol specifications. An example illustrating the use of the proposed mechanisms is presented in Section 6, while Section 7 offers some concluding discussion.

2 Performance and Modularity

Clark and Tennenhouse [3] analyzed performance bottlenecks in protocol processing, and proposed some architectural principles in support of high performance. One theme of their work is that the architecture should leave the protocol implementator with as many options as possible. In particular, constraints on the order in which protocols handle an incoming message should be minimized to the extent possible. Conventional layered architectures require that each data unit be processed by the protocols in a particular order, even when some functions of various layers are logically independent. The principle of *integrated layer processing* says that "the different functions [should be] 'next to' each other, not 'on top of', to the extent possible" [3]. This lets independent functions be performed concurrently, and permits optimization of functions that require handling of user data (e.g. checksums, presentation encoding) by placing them all within a single loop, so that each byte of a message need only be fetched from memory once.

The "next to rather than on top of" idea is also present in the HOPS (Horizontally Oriented Protocol Structure) model of Haas [7]. HOPS features a single protocol that provides all functions at and above the transport level; again the emphasis is on allowing necessary functions to be performed in any order or in parallel. Flexibility is achieved by *selective functionality*: the user can select from a predefined set of options the functions required on a per-session basis. Performance is enhanced because the functions are "horizontally" composed, rather than "vertically". Methods of implementing horizontal composition are not discussed in [7].

Although protocol functionality can be logically layered while the implementation is structured differently, Clark and Tennenhouse admit that integrated layer processing may lead to complex designs and a multiplicity of different customized implementations. Unfortunately, this is just what we would like to avoid, as it increases the cost of porting a single protocol to a new system.

In contrast, the work of O'Malley and Peterson [11] emphasizes the benefits of modularity and flexible composition by taking the concept of layering to an extreme. They propose to implement services by composing various *microprotocols*, each of which performs a single function in a straightforward way. Different combinations of microprotocols provide different services, or the same services over different media. Composition of microprotocols is supported in this architecture by having all protocols implement the same basic set of interface functions. Instead of each protocol being designed to sit "on top of" a specific protocol or layer—as TCP is designed to use IP, for example, or the OSI Session Layer is designed to use the Transport Layer service—all protocols expect the same features from their neighbors in the stack. This approach also aids in development of new microprotocols: the interface to the protocol is fixed and known in advance, so libraries and templates of standard routines can be developed.

To enhance performance, the architecture also supports dynamic inclusion and exclusion of protocols on a per-message basis. In conventional architectures, each message passes through each layer in the stack, whether it needs to or not. So, for example, a layer responsible for fragmenting large packets into smaller ones must handle every packet, including those small enough not to need fragmenting; obviously this may hurt performance. Dynamic inclusion and exclusion allows packets that do not need to be handled by a protocol to bypass it altogether; instead of a linear stack, the protocols form a flowchart-like graph structure. Two mechanisms support dynamic inclusion and exclusion. *Virtual protocols* correspond to the decision points in the flowchart; they select the next protocol to handle a message as it travels down through the protocol stack. On the incoming side, each protocol can determine the identity of the next protocol to process the incoming message by looking at the first few bytes of the next header (located at the beginning of the "data" portion of the message). A *metaprotocol* ensures that all headers are the same length and include a protocol identifier in the first few bytes.

O'Malley and Peterson's architecture uses conventional layering as a composition paradigm: an outgoing message passes through a linear sequence of protocol modules, each of which adds header information to the data unit it received from above. Although the architecture does not constrain an individual microprotocol to a particular position in the stack, the method of composition nevertheless forces the implementation to process the protocol functions in strictly linear order.

We would like to synthesize the foregoing ideas, and identify architectural features that support *modular*

implementations while avoiding unnecessary ordering constraints and other performance impediments. As a step toward this kind of modularity, in the next section we next consider how data flows between protocol modules in an implementation.

3 A Model of Protocol Function

We focus on protocol functions required in *end systems* that send and receive data via a high-performance communication substrate, such as ATM. Thus we are dealing with protocol functions at the transport level and higher. The service provided by the (network level) substrate is unspecified, but is assumed to be an adequate foundation for a variety of communication services.

The term *protocol entity* will be used to refer to the abstract peers of the protocol architecture, i.e. the objects of standard specifications, while the term *protocol module* (PM) refers to the concrete object that implements the behavior defined by the protocol specification. We would like for protocol modules to have well-defined interfaces that minimize reliance upon architectural characteristics.

Our model is based on a view of protocol entities as *transducers*, which take in information in one form and emit information in a possibly different form. Each protocol module does some or all of the following:

- Perform transformations (e.g. presentation encoding) on user data.
- Update local state information in response to control information received from the user, other local PMs, and/or the remote peer.
- Generate control information to be forwarded to the user and/or other local PMs, based on local state plus control information received from local PMs and/or the remote peer.
- Generate control information intended for the remote peer, based on local state and received control information. Note that such *header* information may or may not be attached to user data for transmission over the substrate.

Thus, three types of information may flow across the interfaces of a protocol module: local control information going to/from local modules; remote control (header) information going to/from the module's peer entity; and user data.

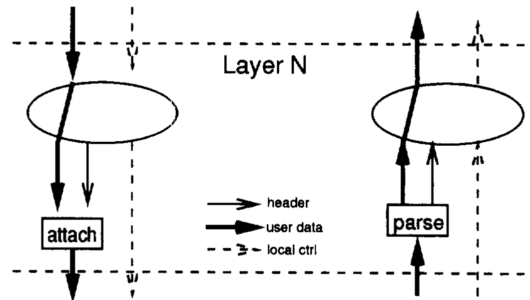

Figure 1: Conventional Layer Implementation

3.1 Conventional Layering

The standard method of composing functions in a protocol architecture is through *layered encapsulation.* That is, the layer N protocol peers exchange control information via the "channel" implemented by layer $N-1$ and thus implement a more powerful channel, which in turn is used by the layer $N+1$ peers. Layer $N-1$ does not—indeed, cannot—distinguish between header information and user data flowing across the interface it provides to the higher layer. In fact, *only* data and local control information (e.g., addresses) can flow across layer interfaces in a traditional implementation. Unfortunately, this method of composition forces an incoming message to be processed sequentially by the layers: a peer entity cannot locate its own header in an incoming message until after the lower-level peers have stripped theirs and processed the message.

The flow of different types of information in a conventional layer implementation is shown in Figure 1. (The three arrow types represent data, local control and header information, and ovals represent protocol modules). After a segment of outgoing data has been processed and a header created, the two are combined to form the data unit passed to the next layer (see the box labeled "attach"). When an incoming message comes up from below, the header is first parsed (the box labeled "parse") and any appropriate per-session information is retrieved.

3.2 Protocol Glue

To maximize interoperability, we need to minimize the assumptions about the architecture embodied in protocol specifications. Therefore we need to carefully separate aspects of protocols that are externally

visible (i.e., affect headers and data) from those that deal with "infrastructure". Included among the latter are those functions that shunt information among protocol modules, such as the "attach" and "parse" functions depicted in Figure 1. We refer to these as *glue functions*.

The "virtual protocols" of O'Malley and Peterson's architecture [11] exemplify the concept of a glue function. These "protocols" determine which microprotocol modules need to handle an individual message at send time. They do not generate headers, nor do they directly affect interoperability. Rather, they are part of the infrastructure, affecting the way actual microprotocols are composed.

Morpheus, a specialized language for implementing protocols [1], provides another example. In Morpheus, built-in objects and functions are available to save the programmer from having to deal with routine matters of infrastructure like header parsing; standard implementations of these glue functions are provided by the compiler. However, protocol implementations written in Morpheus depend on this specific set of glue functions (including conventional layered composition) and thus are portable only among systems that provide the same glue.

Although protocol glue is not typically recognized as such in conventional stack architectures, it is nevertheless present, as we have seen. Clearly more horizontal protocol composition will also require glue functions, e.g. for synchronization of parallel PMs. Our basic approach to architecture-independent modularity will use glue functions to hide differences between composition methods. By supplying the appropriate glue, the same protocol entities can be implemented in both horizontal and vertical architectures. In the next two sections we consider mechanisms for accomplishing this.

4 Explicit Metaheaders

To support horizontal protocol composition, it must be possible for headers of different protocols to be parsed and routed (via protocol glue) to their proper PMs all at once, or in parallel. It should also be possible, however, to process the same header information sequentially, as in a conventional stack. The metaprotocol in O'Malley and Peterson's architecture ensures that all headers are the same length. This is helpful but not quite sufficient, because there is no way to determine, a priori, how many headers are contained in a data unit. One solution is to prefix each message with a single byte indicating the number of headers

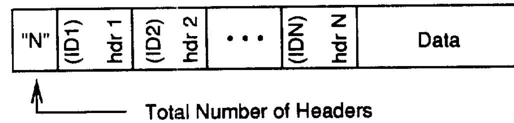

Figure 2: One-byte Metaheader

it contains, as shown in Figure 2. More generally, we can define a fixed-format *metaheader*, which contains sufficient information to permit the protocol headers on a data unit to be located all at once.

We would like for this metaheader to be easily constructible whether an outgoing message is processed sequentially or in parallel. In a parallel implementation, with the scheme shown above (i.e., a single-byte metaheader indicating the number of fixed-length headers, each beginning with a protocol ID), a protocol glue function can collect all the headers and attach them to the data message (if any) along with the header-count byte. For an incoming message, another glue function simply counts off the indicated number of headers, looks at their IDs, and routes each to the proper protocol module.

A vertical implementation (one in which header and data are treated as data when crossing layer boundaries) can add a metaheader to an outgoing data unit by having the "attach" function in each layer remove the metaheader from the outgoing data, attach the layer header, then re-attach the incremented metaheader count. A vertical implementation can simply throw away the metaheader of an incoming data unit, and then process it as a normal message.

Various alternative metaheader formats allowing variable-length protocol headers, more efficient parsing, etc. are easily imagined. Moreover, due to their syntactic nature, it should be straightforward to translate between any two header metaprotocols that have the characteristics described above. Thus, it is not important which header metaprotocol an architecture uses, only that it includes one with the needed properties.

Note also that the use of metaheaders eliminates some performance problems associated with *layered multiplexing*, because it enables per-session state information to be retrieved all at once [6].

5 Generic Interfaces

With the aim of isolating architectural constraints within protocol glue to the extent possible, we propose to specify protocols using a small number of *generic interfaces*. These simple interfaces are classed according to the type of information flowing across them (as discussed in Section 3): incoming or outgoing user data, remote control (header), and local control. By maintaining separation of header, data, and local control information, these interfaces allow protocol functions to be isolated from details of the infrastructure.

We define five classes of generic interface, as follows. A *Data Outgoing* (DO) interface appears wherever user data goes from one module to another on its way toward the communication substrate. A *Data Incoming* (DI) interface passes user data in the other direction. *Header Outgoing* and *Header Incoming* interfaces (HO and HI, respectively) are the analogues of DO and DI for headers. All interfaces pass local control and synchronization information in addition to data or headers. The *Control-Only* (CTL) interface passes only this type of information. (An example of local control information would be a pointer to per-session state information.) Each interface is oriented: one side *offers*, or exports the interface, while the other side *uses*, or imports it.

Any particular protocol module or component will offer and/or use some combination of these interfaces. As a simple example, consider the sending entity of a data transfer protocol using positive acknowledgements. The sending peer attaches a sequence number to each data unit transmitted, and keeps retransmitting the data unit until an acknowledgement containing that sequence number is received. The sending PM (Figure 3) *offers* an instance of the Data Outgoing interface, across which the data units flow; it *uses* an instance each of the Data Outgoing, Header Outgoing, and Header Incoming interfaces. (The arrows in the figures indicate orientation of the interface rather than the direction of data flow: the arrow points toward the *offerer* of the interface and away from the *user*.)

An example of an interface specification is given in Figure 4 for the DO interface; local control info has been omitted for brevity. The *events* of the interface are controlled either by its *user* or *offerer*. A state transition system defines the allowed sequences of event occurrences: associated with each event is a state transition, specified here using a guarded command notation. A transition of the form $b \rightarrow x := e$ has the following meaning: at any state where b is true the event may occur. If it occurs, all the expressions

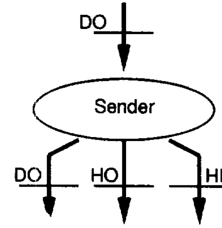

Figure 3: Interfaces of Peer Module

Variables:
x-b: sequence of octet, initially $\perp$;
x-*full*: boolean, initially *false*;
User-controlled Events:
x-*put*(s): $\neg x$-*full* $\rightarrow$ x-*full*, x-b := *true*, s;
Offerer-controlled Events:
x-*ack*: x-*full* $\rightarrow$ x-*full* := *false*;

Figure 4: Data Outgoing Interface Specification

in the list e are evaluated, and the value of each is assigned to the corresponding variable in the list x. The symbol $\perp$ denotes the empty sequence.

As is clear from Figure 4, the generic interface specification says nothing about the *functional* semantics of the protocol entity that offers or uses it. It is analogous to a declaration of the standard procedures imported and exported by protocols in the x-kernel [8], and indeed it should be defined in such a way that the interface can (but need not) be implemented by a procedure call-return mechanism. Details about the semantic relationships among the inputs and outputs of the protocol entity are provided by the protocol specification itself, which includes and refines the generic interface specifications.

An example of such a specification is given in Figure 5 for a simplified entity that applies an encoding function f to segments of user data. The entity *offers* one instance of the DO interface (*up-put*, *up-ack*) and *uses* another (*dn-put*, *dn-ack*). The entity first reads the data segment to be encoded from the *"up"* interface and appends it to a queue of segments; eventually the encoding function is applied to the segment, the local state v is updated, and the result is written to the *"dn"* interface.

The main point of this example is that the specification admits rather different implementation

Variables:
up-b,*dn-b*: sequence of octet, initially $\perp$;
up-full,*dn-full*,*copied*: boolean, initially *false*;
q: sequence of sequence of octet, initially $\perp$;
v: control state;
Input Events:
up-put(s): $\neg$*up-full* $\rightarrow$ *up-full*, *up-b* := *true*, s;
dn-ack: *dn-full* $\rightarrow$ *dn-full* := *false*;
Output Events:
dn-put(s): $\neg dn\text{-}full \wedge q \neq \perp \wedge s = f(hd(q), v) \rightarrow$
$q, v, dn\text{-}full, dn\text{-}b := tl(q), update(hd(q), v), true, s$;
up-ack: *up-full* $\wedge$ *copied* $\rightarrow$
up-full, *copied* := *false*, *false*;
Internal Events:
copy : *up-full* $\wedge$ $\neg$*copied* $\rightarrow$
$q, copied := q \circ \langle up\text{-}b \rangle, true$;

Figure 5: Example Module Specification

approaches,[1] assuming the encoding function f and the state *update* function distribute over concatenation, i.e. if the following hold for any octet sequences s and s', and any value v of the state (where $\circ$ denotes concatenation):

$$\begin{aligned} f(s \circ s', v) &= f(s, v) \circ f(s', update(s, v)) \\ update(s \circ s', v) &= update(s', update(s, v)) \end{aligned}$$

In view of these properties, a pipelined implementation, in which data are passed one byte at a time and interface events occur in the order *up-put*, *up-ack*, *dn-put*, *dn-ack*, produces the same output as a procedure-call implementation, in which an entire segment of data is processed at once and events occur in the order *up-put*, *dn-put*, *dn-ack*, *up-ack*.

The Lam-Shankar theory of interfaces and modules [10] provides a framework for defining and reasoning about interfaces and protocol entity specifications. The theory enables reasoning about the functions implemented by composite systems based upon their individual components, and supports a notion of property-preserving refinement of modules [2]. An example of its potential use would be in showing that a composite of several PMs provides the same function as an integrated implementation.

[1]Informally, a correct implementation generates only sequences of event occurrences allowed by the specification. An event occurrence may correspond to, e.g., a procedure call/return or a handshake between processors.

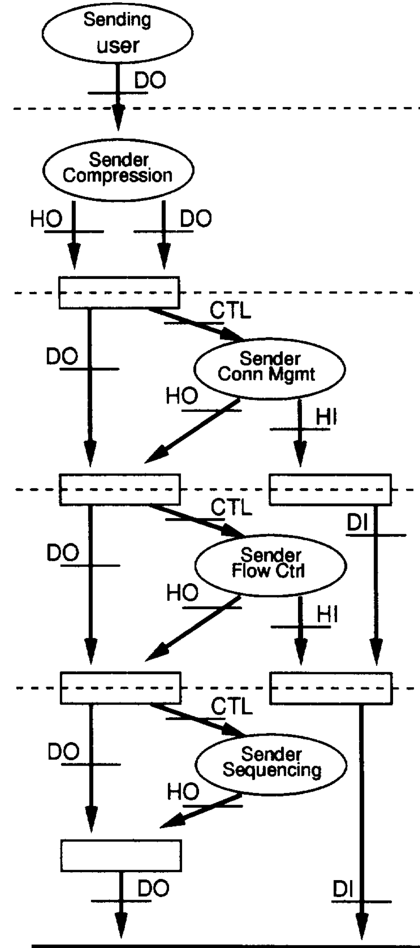

Figure 6: Vertical Sender Implementation

6 An Example

In this section, we present an example showing how the proposed mechanisms can reduce the requirements for interoperability to support for the same set of protocols. Twp implementations are illustrated; one (Figure 6) uses traditional vertical composition while the other (Figure 7) supports more horizontal composition. We claim that these two implementations should be able to interoperate.

The example application involves transfer of bulk data from a sending end system to a receiving system using four protocol functions: *connection management*, *data compression*, *sequencing*, and *flow con-*

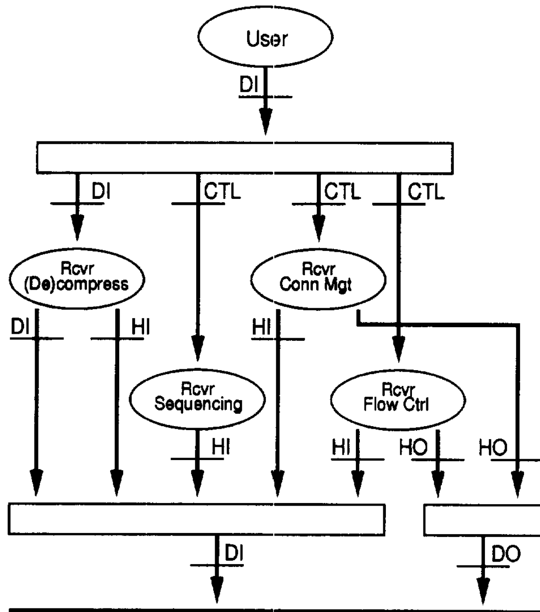

Figure 7: Horizontal Receiver Implementation

functions (connection management and flow control) are assumed to require control information to be sent between peers in both directions; in the other two, headers go from the sender to the receiver only. (Sequencing does not require feedback from the receiver because it does not guarantee that every packet is delivered, only that they are delivered in correct order.)

To illustrate how the mechanisms described above permits the same protocol modules to be composed in different ways, the implementation on the sending side (Figure 6) has a traditional layered structure (with the addition of metaheaders) while on the receiving side the protocol functions are horizontally composed, and can execute in parallel (Figure 7). The interface classes offered and used by protocol modules (ovals) are indicated; boxes represent glue components. Note the regularity of the vertical sender implementation: each layer has a similar structure, with glue components attaching and detaching headers (and metaheaders). In the receiver, the bottom-left glue box parses incoming metaheaders and distributes header and data information; the bottom-right glue box combines headers and metaheader to form an outgoing message. Both of these glue functions are more or less protocol-independent, being concerned mainly with the sizes of the various headers.

7 Discussion

This work is based on the following observations:

- New protocol architectures are on the way.
- All existing architectures require communicating systems to support not only the same protocols, but also the same architectures—in particular, the same method of composition.
- Because there will always be systems that support different architectures but still need to communicate, it is desirable to reduce the requirements for interoperability (above the network layer) so that implementing the same protocols is sufficient.
- Removing architectural constraints on protocols is consistent with implementation structures that support high performance.

Our ultimate goal is development of "interoperability-friendly" protocol architectures that encourage modular, portable protocol implementations without sacrificing performance; this paper is a step toward that goal. We have identified two potential features of such architectures: metaheaders, which permit the various headers attached to a message to be parsed and processed in parallel; and specifications based on generic interfaces, which help separate protocol functions from architecture-specific glue functions. It appears that these mechanisms support interoperability among both vertical and horizontal implementations.

The ideas presented here were described in terms of an architecture—inspired by the one described in [11]—featuring a variety of modular protocols that can be combined to form all kinds of services. The distinction between such an architecture and a single protocol with selective functionality (as in HOPS [7]) is probably mostly a matter of viewpoint. Because it suggests an *extendible* framework, we prefer the architectural view. Alternatively, this paper can be viewed as a step toward a HOPS implementation.

One issue not dealt with here is the run-time instantiation of the composite protocol functions needed by users. The service requested by the user typically determines which protocols are involved in a session. Because only a limited number of protocol configurations may be implemented on any one end system, some means is required to negotiate and configure the required protocols. One possibility is for the protocol configuration to be encoded in the (single) session

identifier. For a more complete discussion of the problem of run-time determination of a remote system's protocol configuration, see [4].

Finally, it should be noted that horizontal composition has a cost in terms of reasoning about protocol correctness. One motivation for grouping functions in layers is that it allows them to be abstactly represented as a monolithic channel, which simplifies proving correctness of the protocols that use them. A non-layered composition mechanism may make it harder to concisely describe the aggregate service provided by a group of protocol functions, and so complicate reasoning about the correctness of simple pairs of peer entities. Note, however, that peers in a conventional protocol often implement several functions at once; reasoning techniques such as projection [9], intended for complex protocols, may be useful for horizontally-composed protocols as well. In any case, maximizing the orthogonality of protocol functions, which allows more parallel and integrated implementations, also simplifies correctness proofs.

References

[1] M. B. Abbot and L. L. Peterson. A language-based approach to protocol implementation. In *Proceedings ACM SIGCOMM '92, Baltimore, MD*, pages 27–38, 1992.

[2] K. L. Calvert. Module composition and refinement with applications to protocol conversion. In *Proceedings XII Symposium on Protocol Specification, Testing, and Verification, Orlando, Florida*. North-Holland, June 1992.

[3] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings ACM SIGCOMM '90 Symposium, Philadelphia*, pages 200–208, 1990.

[4] R. J. Clark, M. H. Ammar, and K. L. Calvert. Multi-protocol architectures as a paradigm for achieving inter-operability. In *Proceedings IEEE INFOCOM '93, San Francisco*, March 1993.

[5] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica. Is layering harmful? *IEEE Network*, 6(1):26–35, January 1992.

[6] D. C. Feldmeier. Multiplexing issues in communication system design. In *Proceedings ACM SIGCOMM '90 Symposium, Philadelphia*, pages 209–219, September 1990.

[7] Z. Haas. A protocol structure for high-speed communication over broadband ISDN. *IEEE Network*, 5(1):64–70, January 1991.

[8] N. Hutchinson and L. Peterson. Design of the *x*-kernel. In *Proceedings ACM SIGCOMM '88 Symposium, Stanford, CA*, pages 65–75, 1988.

[9] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, July 1984.

[10] S. S. Lam and A. U. Shankar. Understanding interfaces. In *Proceedings of the Fourth International Conference on Formal Description Techniques (FORTE), Sydney, Australia*, November 1991.

[11] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[12] D. L. Tennenhouse. Layered multiplexing considered harmful. In H. Rudin and R. Williamson, editors, *IFIP Workshop on Protocols for High-Speed Networks*. Elsevier, May 1989.