# Architectural Concepts in Implementation of End-system Protocols for High Performance Communications

K. Ravindran*
Department of Computer Science
The City University of New York
New York, NY 10031,U.S.A
ravi@cs-mail.engr.ccny.cuny.edu

G. Singh
Department of Computing and Information Sciences
Kansas State University,
Manhattan, KS 66506, U.S.A
singh@cis.ksu.edu

C.M. Woodside
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario K1S 5B6, Canada
murray.woodside@sce.carleton.ca

## Abstract

*The paper presents a functional view of end-system protocol implementations whereby the protocol is decomposed into multiple functions, each causing a change in the protocol state. This view makes the interactions and the relationships among the various functional modules explicit. In terms of this view, currently prevalent architectural optimizations for performance improvement (such as 'parallel executions' and 'integrated layer processing') can be easily described as a set of control flow relationships among modules. If a protocol implementation is analyzed using our functional model, the possible architectural optimizations in the protocol can be easily identified and implemented without violating correctness. Thus, our approach can be used to optimize existing implementations by casting the underlying protocols in our framework, and it is particularly useful in developing implementations for new protocols.*

## 1. Introduction

The search for generality, flexibility and standardization has led to bulky implementations of end-systems. Examples are the TCP and the ISO TP4 based transport systems. The implementations of such systems often conform to the OSI layered architecture. However, the slowness of execution of the protocol implementations, which is essentially due to sequential processing of the complex protocol procedures for each data unit in various layers, is becoming a limiting factor in some emerging applications which require high bandwidth and large volume data exchanges between various application devices through a high speed backbone network. For example, the transfer of images and live video information may involve transporting hundreds of gigabytes of data at high rates, typically in the range of 150 to 200 *mbps* [1]. Such a capability cannot be met by existing protocol implementation structures, which typically support transfer rates ranging between 750 *kbps* to 6 *mbps* [2, 3]. This warrants high performance implementations of end-systems that can provide high transfer rates meeting the application needs, limited only by the network speeds.

The various processing activities on a application-specific data unit (or, packet) such as scheduling control, multiplexing and presentation level processing of data are part of the end-system protocol. The protocol can also include lower level functions such as rate control and error recovery on data. Figure 1 illustrates the placement of these functions in an end-system node with respect to application entities and the backbone transport network attached to this node.

The ways in which various functions influence the overall performance of end-systems are often difficult to be analyzed in a systematic way and generalized for broad usage. This difficulty can often obscure many performance engineering aspects that may be inherently possible. For instance, the communication level processing of a video picture data can proceed in parallel with the compression/decompression of this data, provided the presentation and communication activities on the video data are carefully separated. This parallelism can be obscured if, for instance, a conventional layered implementation of various protocol

*Part of this work was performed when the author was at Kansas State University.



0-8186-7453-9/96 $05.00 © 1996 IEEE

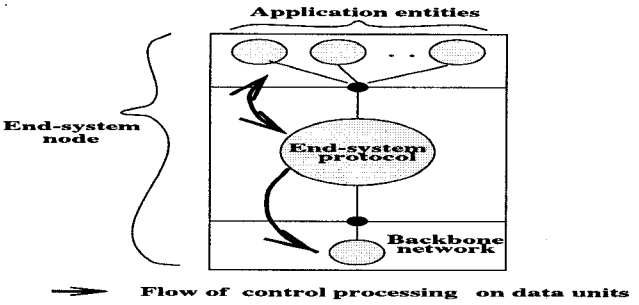


Figure 1: External view of end-system node architecture

functions is employed. Thus, it is highly desirable to identify the generic principles that may be useful in structuring end-system implementations for high performance. Towards this end, the paper navigates through:

- Representative end-systems to extract the architectural concepts entwined into implementation details;
- The evolving application domains (such as multimedia communications) to identify newer requirements that can be translated to architectural elements of an end-system implementation.

These concepts and elements can then be employed in a more judicious manner, as required, for tailoring a target end-system in future implementations and/or for re-engineering the existing end-systems.

Our approach is to decompose a protocol into a set of *orthogonal* procedures, i.e., procedures that do not interact with one another, and realize them by separate functional modules. See Figure 2. In multimedia communications for instance, the transport functions for the data stream of each media may be viewed as a distinct protocol handling the media-specific delivery guarantees (e.g., acceptable data delays). From an implementation standpoint, the decomposition of a protocol will project the process implementing the protocol into a subspace of asynchronous processes (possibly executing on different processors) coordinating with one another to share the protocol state. And these processes are constrained by protocol-specific conditions, and often, implementation-specific optimizations.

To isolate implementation-specific aspects from an end-system structure, the paper identifies a *functional view* of end-system protocols, wherein a protocol is decomposed into orthogonal functions, with each function executing a distinct task as determined by the transport header of data and the results of these functions are composed to generate a state change in the protocol. This view allows us to analyze the fine-grained interactions among various functions in an end-system protocol. The analysis in turn reveals the necessary architectural elements of a high performance target end-system. The techniques discussed in the paper are:

- *Parallel execution* of protocol functions to exploit the inherent parallelism among communication activities.
- *Look-ahead processing* of protocol functions to exploit the predictability of communication flows;
- *Function bypassing* to eliminate redundant processing and/or reduce processing requirements of one or more protocol functions, wherever feasible[1].

These engineering techniques, when incorporated into a protocol implementation architecture, serve to improve the end-to-end data transfer performance (i.e., increased throughput rate and reduced latency)[2]. We formalize these techniques to allow a better insight into various architectural optimizations possible in end-system implementations. More specifically, these techniques — as identified in the past by various research groups through the 'hard way' — can be systematically analyzed in terms of our function-based communication model. This in turn benefits future implementations of high performance end-systems in the form of a systematic exposition of possible optimizations.

To formulate various elements of the communication model for analyzing the engineering techniques, we draw corroborative observations from 3 different prototype end-systems: the multimedia data delivery system built at Kansas State University [5], the OSI-layered protocols implemented at Carleton University [6, 7], and the blast protocols implemented elsewhere by other researchers [2, 3]. In retrospect, we show how the architectural techniques embodied in these end-systems — in one form or the other — can be viewed through our model. The paper presents architectural concepts in a generalized manner so that they can be adopted for use in any target system.

The paper is organized as follows: Section 2 presents a 'functional view' of end-system protocol implementations. Section 3 presents the support mechanisms required in end-systems to support this view. Based on the functional view, sections 4–6 extract the generic architectural concepts. Section 8 concludes the paper.

## 2. 'Function-based' view of end-systems

This section deals with the basics necessary to evolve the architectural engineering concepts that may be useful in structuring high performance end-system protocols. This is based on decomposition of protocol functions and analyzing the interactions among the component functions.

We assume that the headers affixed to data packets activate various functions in the end-system. We focus primar-

[1] The 'bypass' concept arises from an *integration* of protocol functions by weakening the interfaces between layers that realize these functions (i.e., have a protocol function 'snoop' into the activities of other functions to optimize its behavior). [4] refers to this principle as 'integrated layer processing'.

[2] The performance improvement techniques are 'non-intrusive' in that the functional behavior of 'native' implementation of a protocol, as derived from its specifications, is not affected.

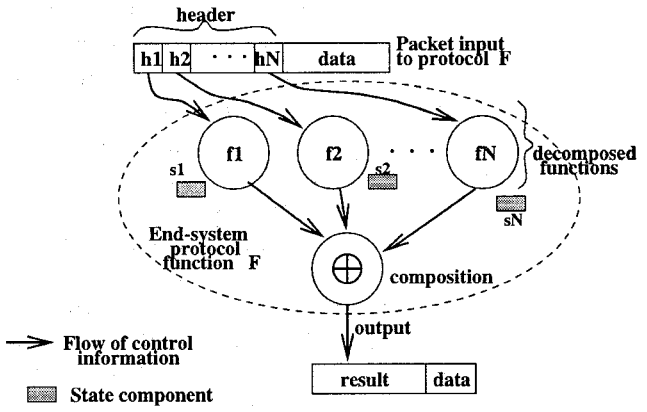


Figure 2: Protocol functions and packet header elements

ily on the control path of a data packet, i.e., the set of functions through which the packet needs to flow for processing.

## 2.1 Header-driven protocol decomposition

Consider the data exchanged between application entities, such as multimedia devices and file system processes, in the form of packets. A protocol is a set of functions invoked by the transport header defined for a data packet. Let $\mathbf{S}_-$ and $\mathbf{S}_+$ be the protocol states before and after processing a packet header $\mathbf{H}$. Then

$$\mathbf{S}_+ = \mathbf{F}(\mathbf{S}_-, \mathbf{H}), \tag{1}$$

where $\mathbf{F}$ is the protocol function. The header $\mathbf{H}$ may be viewed as specifying a set of attributes $h_1, h_2, \ldots, h_N$, and $\mathbf{F}$ operates on each of the attributes to generate a transition from the current state $\mathbf{S}_-$ to the new state $\mathbf{S}_+$. For instance, the loss control and delay control information of various streams are attributes in a multimedia end-to-end transport header. Thus, relation (1) may be written as

$$\mathbf{S}_+ = f_1(h_1, s_{1\_}) \oplus \ldots \oplus f_N(h_N, s_{N\_}), \tag{2}$$

where $s_{j\_}$ and $s_{j+}$ represent the protocol state component before and after processing $h_j$ respectively (for $j = 1, 2, \ldots, N$) and '$\oplus$' is a composition operator to generate a resulting state for the protocol. The decomposed functions $f_1, f_2, \ldots, f_N$ can execute in parallel, up to the point where they generate the results as required by the '$\oplus$' function. The latter then combines the results in a problem-specific manner. See Figure 2. As an example, packet 'delivery schedule' is a state in the delay control procedure of a transport protocol while packet 'discard schedule' is a state in the loss control procedure. And state composition is to impose the constraint that a packet be delivered only if it is not in the 'discard schedule'. The enforcement of such constraints is embodied into the composition function '$\oplus$' in the form of flow of control interactions among component functions.

## 2.2 Parameterizable protocol executions

Protocol functions may be described by (attribute,value) pairs where 'attribute' refers to a function by name and 'value' is the parameter supplied to an invocation of this function. Each header $h_j$ in a packet is an implicit attribute, causing invocation of the appropriate function $f_j$ in the end-system protocol. The content of header field is the value parameterizing this invocation. For instance, the header field carrying the stream id may be represented as (STRM,$x$), where STRM refers to the function that maps stream id $x$ to node-specific end-point address of the destination, say by a table lookup. The state may be a count of the number of packets received so far for stream $x$. Other examples of protocol attributes are packet priority and presentation encoding (such as JPEG/MPEG for video streams).

Many existing protocols use a 'position-oriented syntax' of packet header whereby a field is identified by its offset relative to the beginning of packet header. Recently, proposals have been made for 'position-independent' syntax based on '(type,length,value)' encoding of header elements, which makes the parsing of header to determine the protocol functions easier and more efficient [10]. Regardless of the syntactical structure employed, the function referenced by a header field is ready to execute once this part of header is available to the protocol and the control state resulting from the processing of any other field(s) in the header, as may be necessary, is computed. In the earlier example, the 'loss control scheduler' executes as soon as the priority information in packet header arrives and the index to local information on cumulative packet loss for (STRM,$x$) is available.

## 2.3 Protocol interconnections

We view an end-system as consisting of the set of protocol modules $f_1, f_2, \ldots, f_N$ interconnected with one another and performing specific tasks. The interconnection path depicts the control flow across various modules during processing of a data packet, based on the ordering relationship among their executions that reflects the semantics of packet processing. Typically, the results produced by a module $f_j$ can be processed only in the context of results from another module $f_i$, for some $i, j \in \{1, 2, \ldots, N\}$. Expanding the earlier example of loss and delay control on packets, $f_i$ and $f_j$ may implement the corresponding procedures, with the 'delivery schedule' from $f_j$ processed in the context of the 'discard schedule' from $f_i$. Such an ordering relationship between the functions $f_i$ and $f_j$, denoted as $f_i \rightarrow f_j$, depicts that the results of $f_i$ need to be processed before that of $f_j$. This '$\rightarrow$' relation is spatially representable by a directed edge from vertex $f_i$ to vertex $f_j$ in a graph. See Figure 3-(a). The composition function '$\oplus$' may thus be viewed as enforcing the '$\rightarrow$' ordering relations prescribed among the functions $f_1, f_2, \ldots, f_N$ as part of the execution of $\mathbf{F}$, with the extent of parallelism within $\mathbf{F}$ determined therefrom.

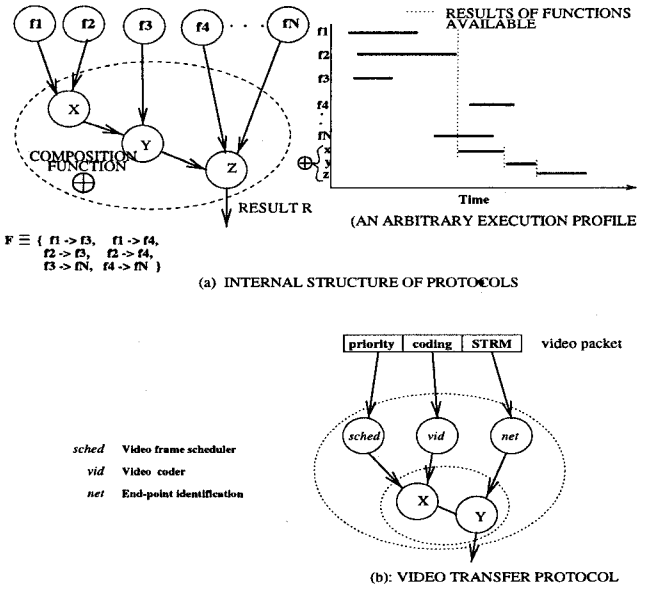


Figure 3: Functional view of protocols in end-system node

As can be seen, the end-system structure depicts a 'dataflow programming' style of capturing the interactions between various protocol modules, with each module instantiated by a header supplied parameter value. And the functional relationships between these modules are encoded in the control flows along inter-module paths[3]. Such a view of end-system node structure aligns with the 'dynamic protocol architectures' proposed in [12].

In many existing protocols, our functional view is only partly apparent in that the header-based decomposition of protocol functions and the isolation of function return values from internal states are not directly incorporated in protocol formulations. In this sense, the functional view may serve as only a 'guideline' in stepping through existing protocols, but offer a fundamental approach in formulating new protocols.

We now describe the support mechanisms necessary to implement the functional view of end-system protocols.

# 3 Architectural support mechanisms

This section deals with the mechanisms necessary to support our functional view of protocols.

## 3.1 Data 'cut-through'

A data packet 'visits' all protocol functions to 'solicit' their share of processing. So a packet contains header elements pertaining to each function. When a packet is input to the protocol, its header is made available to all component functions. The packet may thus be viewed as 'cutting through' all functions in an end-system. The 'data cut-through' may be viewed as an analogue in the functional model to the notion of 'application-level framing' of data streams [4], i.e., communication level processing of data in application-specific units (e.g., a video picture frame processed by various functions as a single data unit).

The 'data cut-through' allows delineating the processing activities on a packet across distinct protocol functions (say, separating the communication and presentation level processing of a graphics data unit). And a proper exercising of these functions to process packets can be guided by the functional model discussed in section 2. In a video-based application for example, the compression of a video picture frame and the scheduling control on the frame should both complete before the data can be sent over the network. And these ordering relationships can be enforced by realizing the video compression ($vid$), communication scheduling ($sched$) and network transport ($net$) as distinct protocol functions, and then interconnecting these functions in the form $\{(vid \rightarrow net), (sched \rightarrow net)\}$ (see Figure 3-(b)). Thus our functional view of end-system implementations aligns with current architectural notions.

## 3.2 Efficient synchronization tools

Researchers envisage realization of high performance end-systems on shared memory multiprocessors, with each decomposed protocol function implemented on a separate processor [8, 9]. For example, presentation control procedures on various data streams of a multimedia application can execute on different processors and synchronize themselves at appropriate execution points. The protocol state may be maintained in shared memory for access by various functions during execution. The '$\rightarrow$' relationship among functions may determine the extent of parallelism by forcing their processors to synchronize their access to the state.

Typically, *spin-locks* provide an efficient way of synchronizing the executions on various processors through shared memory [13], whereby a processor waits by 'spinning' on a lock, i.e., by busy-looping until the lock variable in shared memory is set by another processor. And the lock variable is reset once the 'spinning processor' unblocks from the wait. Typical overhead incurred for spin-locks (excluding the 'spinning' time) may be is about 5-10 $\mu$ sec on dedicated RISC multiprocessor platforms. So a desired level of synchronization may be achieved inexpensively so that the benefits of optimized protocol executions can be reflected on protocol performance.

## 3.3 Walk-through of a sample protocol

Consider, as an example, the video transfer protocol in the SUN video conferencing demonstration program. A video data frame is sent and received by the camera and display devices respectively as a set of up to 16 'network packets' (NDU) over UDP-ethernet. We describe this protocol in

[3] A somewhat similar approach has been adopted in [11] to structure the 'user-network interface' in multimedia communication systems.

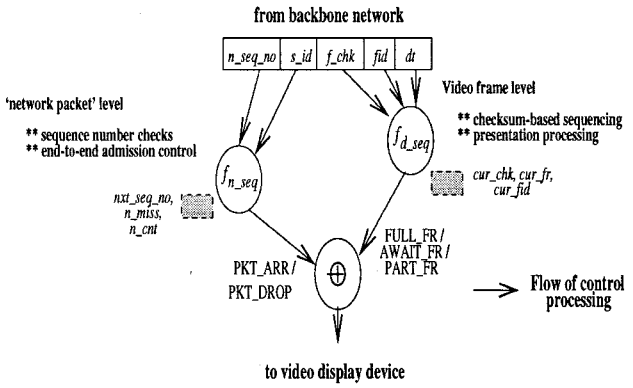


Figure 4: Functional components of video transfer protocol

terms of our functional view. There are 2 sequencing control functions on an incoming NDU: $f_{n\_seq}$ at the network interface layer with the use of sequence numbers to allow end-to-end support for admission control on data streams (say, for congestion avoidance in the network), and $f_{d\_seq}$ at the device layer for frame level checksumming across all NDUs of a frame. For an incoming NDU $pkt$, the invocation of these functions may be represented as:

$$\{\text{PKT\_ARR}, \text{PKT\_DROP}\} = f_{n\_seq}(\{pkt.n\_seq\_no, pkt.s\_id\}, \{nxt\_seq\_no, n\_miss, n\_cnt\})$$

$$\{\text{FULL\_FR}, \text{AWAIT\_FR}, \text{PART\_FR}\} = f_{d\_seq}(\{pkt.f\_chk, pkt.f\_id, pkt.dt\}, \{cur\_chk, cur\_fr, cur\_fid\}),$$

where $nxt\_seq\_no$ is the next sequence number expected by the network interface module, $n\_cnt$ and $n\_miss$ indicate respectively the number of NDUs received so far and missed so far, $cur\_fid$ is the current frame id, $cur\_fr, cur\_chk$ indicate respectively the frame assembled and the accumulated checksum value from NDUs received, and $s\_id, f\_chk, dt$ indicate the stream id, checksum value, device data respectively carried in a NDU. The definitions of $f_{n\_seq}$ and $f_{d\_seq}$ are as illustrated in Figure 4. The '$\oplus$' function uses the results to enforce the ordering relationship $f_{n\_seq} \rightarrow f_{d\_seq}$.

Here, a data packet 'cuts through' $f_{n\_seq}$ and $f_{d\_seq}$ to have these functions produce PKT_ARR/PKT_DROP and FULL_FR/AWAIT_FR/PART_FR respectively. Synchronization may be realized by having $f_{n\_seq}$ 'lock' a shared variable until it computes the result PKT_ARR/PKT_DROP and $f_{d\_seq}$ 'spin' on this variable.

The functional view of end-system protocols and the underlying support mechanisms, as described so far, form the basis for describing the architectural elements of a high performance end-system. These concepts are discussed in the following sections 4-6.

# 4 Parallel execution of protocols

Consider a protocol function **F** made up of sequential processing activities on an incoming packet, i.e., one activity should complete before the next activity can start (e.g., network layer routing table lookup and transport layer stream identification in a OSI 7-layer implementation). In some cases, it is possible to execute some parts of an activity in parallel with its previous activity without violating the sequentiality constraint. How this parallelism can be captured in protocol formulations is described below.

## 4.1 Functional view of parallel execution

Suppose **F** is decomposed into two functions $f_1$ and $f_2$, as governed by the relation: $f_1 \rightarrow f_2$. The decomposition is such that $f_2$ can execute in parallel with $f_1$ up to a certain point and the synchronizer $\oplus$ has the results of $f_1$ processed before that of $f_2$. In other words, packet processing can start simultaneously at $f_1$ and $f_2$, but the completion of **F** will occur only after processing the results of $f_1$ and $f_2$ in that sequence. Such a parallel execution allows a view in which a packet simultaneously 'resides' in $f_1$ and $f_2$. As can be seen, a fine-grained parallelism is possible among modules, even when there is a certain ordering relationship to be maintained in composing their executions.

We illustrate the above view with the video transfer protocol example. The functions $f_{n\_seq}$ and $f_{d\_seq}$ may execute in parallel on a given NDU of a video frame; however, if the NDU assembles into a complete frame, $f_{d\_seq}$ transfers this frame to the video display device only after $f_{n\_seq}$ has completed its processing on the NDU. This synchronization is required to account for the case where $f_{n\_seq}$ decides to drop the NDU as part of an admission control mechanism. The above structure offers a high degree of parallelism between $f_{n\_seq}$ and $f_{d\_seq}$.

## 4.2 Other examples

In the earlier example of delay control and loss control on an incoming data packet, these functions can be executed in parallel up to a point where the packet 'delivery schedule' and 'discard schedule' are generated. Thereupon, the stream controller removes those packets listed in the 'discard schedule' from the 'delivery schedule' and arranges delivery of the remaining packets.

In a data transfer session establishment protocol, the function that estimates the resources needed for sending/receiving data and the function that authenticates the end-point users can execute in parallel. However, the resource allocation can be performed only after the authentication completes.

In general, parallelism in execution is determined by the level of functional decomposition feasible on a protocol.

## 4.3 Pipelining of packets

Consider the decomposition of a protocol **F**, denoted as $\mathbf{F} \equiv \oplus[f_1, f_2, f_3, \ldots]$. It is possible to extend the paral-

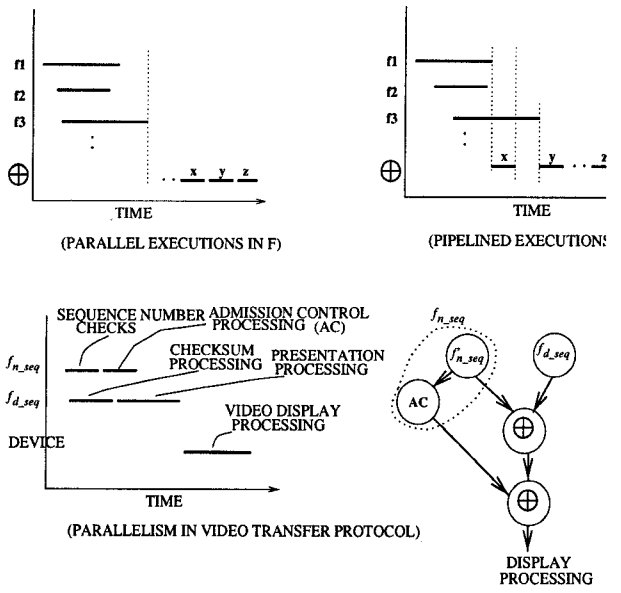


Figure 5: Parallel and pipelined executions in protocols

lelism among $f_1, f_2, f_3, \ldots$ to the internal execution of '$\oplus$'. To illustrate this, consider the relations $\{(f_1 \rightarrow f_3), (f_2 \rightarrow f_3)\}$ shown in Figure 3. Suppose $f_1$ and $f_2$ produce their results before $f_3$ does. A subfunction $x$ of $\oplus$ may process the available results up to the point where the results of $f_3$ are needed. When the latter become available, another subfunction $y$ of $\oplus$ may process the results produced by $x$ and $f_3$. The results generated by $y$ may be used by other subfunctions of $\oplus$. Here, the execution of $x$ overlaps that of $f_3$, and possibly, $y$ overlaps with other functions. Thus pipelining is basically extending the parallelism into the execution of '$\oplus$' function, thereby offering fine-grained parallelism.

In the video transfer protocol described earlier, the function $f_{n\_seq}$ may trigger the execution of $f_{d\_seq}$ right after the internal sequence count variables are updated but before admission control checks are performed. This is because the latter checks on a packet by $f_{n\_seq}$ may not influence the checksum processing on the packet by $f_{d\_seq}$. So the checksum calculations on the packet can take place in parallel at $f_{d\_seq}$ while the sequence checks are done at $f_{n\_seq}$. However, display processing on the video frame to which the packet belongs cannot be started until admission control checks on the stream decide to continue with the processing of this packet. Figure 5 illustrates how pipelining is realized in the video transfer protocol example.

### 4.4 Case study of device-network data transfer

We have studied the pipelining concept in the design of a dedicated multiprocessor based end-system node for sending/receiving of packets from/to multimedia devices over the network interface. Often, the leading part of a packet $p$ may have arrived from the network, and once the local destination device for $p$ has been identified from this part, say, by a table lookup, the device can start processing $p$, even before the remaining part of $p$ has arrived[4]. Effectively, $p$ may be processed by the device, table lookup function and network interface modules in a linear chain (here, 'network($p$) $\rightarrow$ lookup($p$) $\rightarrow$ device($p$)). However, synchronization is needed to avoid one processor over-running the other, which can be achieved by using spin-locks.

Suppose $p$ is an incoming packet processed through shared memory. The network can activate the device to start processing $p$ from shared memory as soon as the device is identified from the header information carried in $p$. Assume 1000-byte packet size, and the network and device speeds as 600 *mbps* and 100 *mbps* respectively. For device table lookup time of 5-6 $\mu sec$ at the link control processor (such as M68030), the device processing can be started when about 40% of $p$ has been received over the link. Likewise, if $p$ is sent from the device to network, the link processor can start the transfer of data over the network after about 84% of $p$ has been filled in the shared memory.

It is possible however that, in some cases, the overhead incurred for the additional synchronization out-weighs the advantages of pipelined executions.

## 5 Look-ahead execution of protocol functions

In certain cases, a protocol function may *predict* the header of an incoming packet and perform some processing even before the packet has arrived (if the processor has slack processing time). Consider, for example, the arrival of a video picture frame segmented into multiple packets by the network. When the first packet of a video frame arrives, the resource manager can predict the arrival of remaining packets of the frame, and hence pre-allocate buffer slots for all these packets. When packets actually arrive, the network module simply places these packets in already-allocated slots. Such a look-ahead processing may improve performance by reducing packet latency and/or increasing throughput rate. This has been demonstrated in many existing protocol implementations [6, 14].

### 5.1 'functional' view of look-ahead processing

The look-ahead processing is possible due to de-lineation of a protocol function invocation from the generation of parameter values for this invocation. In other words, a function invocation does not depend upon how the parameter values are generated: whether from the header of a packet actually received or by fabrication using header prediction. Thus parameterizable function invocation is an architectural support mechanism to enable look-ahead processing of packets.

Performance improvement due to look-ahead processing depends upon the 'prediction range', i.e., how many packets

[4] In many multimedia applications, device level processing of data is often accompanied by a significant amount of presentation processing such as video compression/de-compression and graphic image enhancement.

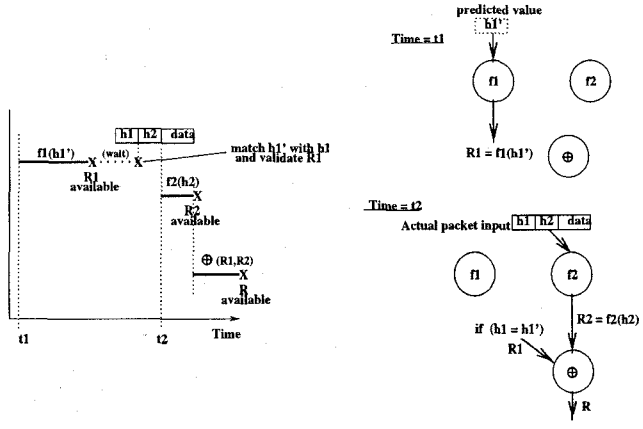


Figure 6: Look-ahead processing in end-system protocols

are expected to arrive next meeting a given criteria, and the amount of processing required on each packet. If any of the next set of packets does not match the prediction (typically, due to network errors), the results accumulated so far by look-ahead processing are discarded and the normal 'upon-arrival packet processing' is resorted to until conditions become amenable again for look-ahead processing[5]. See Figure 6 for an illustration.

In our implementation of video transfer over UDP-ethernet [5], a video picture frame consists of 16 NDUs; so, 15 NDUs of a given stream id are predicted to arrive in sequence next. If a received NDU is not the next expected one, the entire frame is discarded, causing de-allocation of buffer slots for the entire frame. This look-ahead processing is incorporated in the network interface component of video transfer protocol.

### 5.2 Cost implications of look-ahead processing

The confidence level of correct prediction should be high in order to reap the benefits of look-ahead processing. Suppose $k_{pred}$ ($\geq$ 1) is the prediction range. If $T_{tot}$ and $T_{bef}$ are the time durations respectively for complete processing of packet and for processing that can be performed ahead of packet arrival, the effective per-packet processing time $T_{pkt}$ may lie in the range $(T_{tot} - T_{bef}) \leq T_{pkt} \leq (T_{tot} + T_{undo})$, with the actual value depending on the probability $q$ that the next $k_{pred}$ packets match the prediction. Here, $(T_{tot} - T_{bef})$ is the remaining time needed to complete the processing on a packet and $T_{undo}$ is the time to invalidate (or undo) the finished processing on a packet in case of a packet violating the predication. In a window-based transport protocol, $k_{pred}$ may be the window size. Sample values may be $k_{pred} = 6$, $T_{tot} = 5\ msec$, $T_{bef} = 3\ msec$ and $T_{undo} = 1\ msec$; the probability of packet loss in the network can be used as $q$.

The optimized implementation of bulk data transfer protocols in the V-Kernel [3] embodies look-ahead processing based on header prediction. Upon start of a 'packet blast', the protocol expects that the next set of packets arriving over the network will belong to this blast, and hence readies application buffers ahead of time. When data packets actually arrive over the network, they are directly copied to the application space. If the header of a received packet does not match with the expected one, the predictor 'undoes' the data copying by reclaiming the 'garbaged' application buffers, and subjects the packet through normal protocol processing (which may include a recovery by requesting the remote peer to retransmit all packets of the blast). With low packet loss in the network, $q$ is quite high, thereby resulting in high throughput rate. In terms of our model, the blast protocol can be characterized by the following parameters: $k_{pred} = 32$, $T_{tot} \approx 10\ msec$, $T_{bef} \approx 6.25\ msec$, and $T_{undo} \approx 3\ msec$.

Consider the earlier example of buffer pre-allocations for a video frame upon arrival of first NDU of the frame. When one ore more expected NDUs of the frame are not received, the protocol discards other NDUs of the frame and releases the buffer. In our implementation on UDP/IP network [5], $k_{pred} = 15$, $T_{tot} \approx 20\ msec$, $T_{bef} = 5\ msec$, $T_{undo} \approx 4\ msec$, and $q = 0.95$ (for the measured 5% packet loss).

Overall, the formalized notion of look-ahead processing allows exploiting this protocol feature more effectively in an implementation.

## 6 Bypassing of protocol functions

In some parts of protocol execution, complex functions can be replaced by simpler ones that produce the same effect with less computation. For example, collapsing the buffer allocation for individual packets of a frame into a single allocation of a large buffer size achieves the same result with less interactions with 'memory management' functions. Likewise, the complex error and flow control mechanisms in data transfer protocols need not be exercised most of the times. We refer to such a replacement of functions by equivalent but computationally cheaper ones as a 'protocol bypass'. Bypassing can improve protocol performance significantly [3, 6, 14].

### 6.1 'functional' view of protocol bypass

Consider a region of protocol state space bounded by $\mathbf{S_1}$ and $\mathbf{S_2}$ where the dependencies of one or more parts of the state on packet header, as depicted in equation (1), can be eliminated. Accordingly, equation (1) is expressible in the form:

[5] Look-ahead processing may be viewed as an 'operation cache' whereby an operation is performed based on an expected data input, and the results of the operation are simply used when the expected data actually arrives. If the prediction turns out to be incorrect, the cached results are invalidated, and the operation is performed on the actual data.

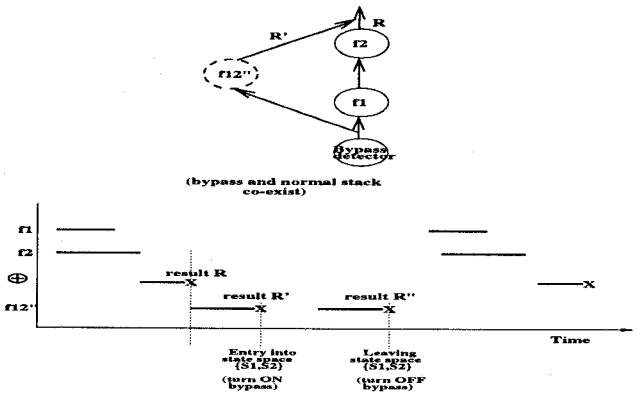


Figure 7: Protocol bypass modules in end-system node

$$\begin{aligned}\mathbf{S}_+ = f_1(h_1, s_{1\_}) \oplus \ldots \\ \oplus f_{(j-1)}(h_{(j-1)}, s_{(j-1)\_}) \\ \oplus f_{(j+k)}(h_{(j+k)}, s_{(j+k)\_}) \\ \ldots \oplus f_N(h_N, s_{N\_}) \\ \text{for } 1 \leq k < N \\ = \mathbf{S}_- \odot \widehat{\mathbf{F}}(\mathbf{H}) \end{aligned} \quad (5)$$

where $\widehat{\mathbf{F}}$ is a function that operates on $\mathbf{H}$ to generate the same state transition as $\mathbf{F}$ does but with the dependency of state components $s_j, \ldots, s_{j+k-1}$ on packet header eliminated and '$\odot$' is a superposition operator. Due to elimination of the dependency on one or more state components, $\widehat{\mathbf{F}}$ is computable at a less cost than $\mathbf{F}$, indicated as $\text{cost}(f_j(h_j, s_j)) \geq \text{cost}(s_j \odot \widehat{f}_j(h_j))|_{j=1,2,\ldots,N}$.

The checking of whether $\mathbf{S}_- \in \{\mathbf{S_1}, \mathbf{S_2}\}$ or otherwise (denoted as SUCC and FAIL respectively) is protocol-specific, and may be carried out by a *bypass detector* (BPD). Since $\mathbf{F}$ needs to examine $\mathbf{S}_-$ anyway to determine its actions, the BPD can be derived from the protocol specific state check mechanisms. With SUCC result, $\widehat{\mathbf{F}}$ may be invoked to process the packet; with FAIL result, $\mathbf{F}$ may be invoked. Accordingly, a bypass-based protocol implementation may be represented as

$$\mathbf{BPD} \rightarrow (\mathbf{F} \vee \widehat{\mathbf{F}})$$

where '$\vee$' depicts the SUCC and FAIL outcomes possible from BPD. With a high probability of SUCC outcome, the bypass-based protocol implementation can incur less processing cost. See Figure 7 for an illustration.

The bypass and normal protocols coexist. Once state transitions into the region of state space $[\mathbf{S_1}, \mathbf{S_2}]$ are detected, the bypass is turned ON and all subsequent packets arriving from network are routed through the bypass module until a transition to outside this region is detected.

### 6.2 Bypass in 'bulk transfer' protocols

Consider an end-to-end communication protocol $\mathcal{P}$ that transfers data from the network to an application. $\mathcal{P}$ may be viewed as consisting of functions $f_{n\_c}$ and $f_{c\_a}$ that copy the data of a packet *dat* from the network to a communication buffer $buf_c$ and from the latter to the application respectively. The case of in-sequence receipt of data packets may be represented as:

$$\begin{aligned} f_{n\_c}((seq_c, buf_c = \\ \text{EMPTY}), s\_no) = (buf_c = \text{FULL}) \\ f_{c\_a}((seq_c = x, buf_c = \text{FULL}), dat) = \\ (seq_c = x + 1, buf_c = \text{EMPTY}), \end{aligned}$$

where $seq_c$ is the sequence number of packet next expected by $f_{n\_c}$ and $s\_no$ is the sequence number of packet received. Here, $\mathcal{P} \equiv f_{n\_c} \rightarrow f_{c\_a}$, with $f_{c\_a}$ executing upon receiving a SUCC result from $f_{n\_c}$, where SUCC indicates that $s\_no$ and $seq_c$ are equal. The case of $s\_no$ and $seq_c$ not being equal (FAIL) initiates a protocol specific recovery by $f_{n\_c}$.

A high performance implementation of $\mathcal{P}$ that avoids data copying into $buf_c$ (such as the 'blast transfer protocol' [3]) may be viewed as consisting of: i) a bypass detector $\text{BPD}(seq_c, s\_no)$ that checks if $s\_no$ and $seq_c$ are equal, and ii) a function $f_{n\_c\_a}((seq_c = x), dat) = (seq_c = x + 1)$. Here, $\mathcal{P} \equiv \text{BPD} \rightarrow f_{n\_c\_a}$, with $f_{n\_c\_a}$ executing upon receiving a SUCC result from BPD. The elimination of $buf_c$ state transitions in $f_{n\_c\_a}$ captures the bypassing of data copy into the communication buffer. The case of FAIL result from BPD will however cause the invocation $f_{n\_c} \rightarrow f_{c\_a}$.

In terms of our model, the composite protocol implementation may be viewed as $\text{BPD} \rightarrow ((f_{n\_c} \rightarrow f_{c\_a}) \vee f_{n\_c\_a})$. Due to the elimination of functional dependency on state $buf_c$ in $f_{n\_c\_a}$ and because $f_{n\_c}$ is at least as expensive as BPD and $f_{c\_a}$ causes the same transitions on $seq_c$ as $f_{n\_c\_a}$ does, the computation $\text{BPD} \rightarrow f_{n\_c\_a}$ can be cheaper than $f_{n\_c} \rightarrow f_{c\_a}$. So with a high probability of SUCC result from BPD, the bypass implementation of $\mathcal{P}$ can yield a better performance.

It is possible that some packets are routed through a bypass module and others through the normal protocol stack. The bypass module itself can execute in parallel with the regular modules that it is bypassing. This parallelism, compounded by the reduced processing time in bypass module, can cause the packets flowing through the bypass path to 'overtake' those sent through the normal path prior to turning ON the bypass. So the interleaving of packet flow events output by the bypass module and the regular modules has to be controlled by appropriate synchronization.

## 7 Conformance of optimized implementation

The native implementation of an end-system consists of a collection of protocol modules with a prescribed interconnection path between them for control flows during execution. An optimized implementation of the end-system employs one or more of the architectural techniques discussed in sections 4-6 to provide an alternative interconnection structure and/or prescription of the protocol modules in

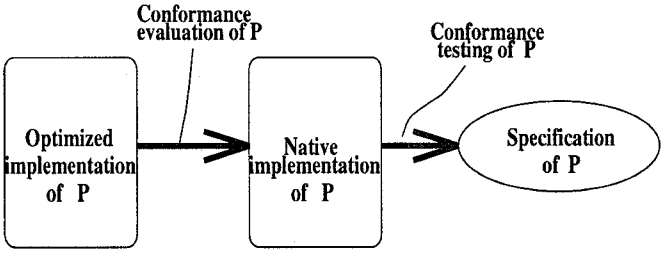


Figure 8: Scope of conformance for optimized implementation

such a way that the external interface to the end-system as a whole remains the same as with the native implementation (see Figure 8)[6]. This conformance is implicit (i.e., comes for 'free') if the implementor correctly models native implementations using our approach, and then incorporates optimizations, as per the procedures suggested.

## 8 Conclusions

We have presented a framework for specifying (or viewing) the protocol at an end-system as a set of decomposable functions, each causing a change in the protocol state. This view makes the interactions and the relationships among the various functional modules explicit. For some of these interactions/relationships, we have identified architectural elements which can potentially improve performance. Once the interactions between modules are made explicit, one can use these architectural elements in a mechanical way. Traditionally, optimized implementations have been obtained in an ad-hoc fashion, wherein each implementation effort is coded and then analyzed to determine whether it results in a performance improvement. On the other hand, if protocol implementations are analyzed using the functional view proposed, optimizations such as exploiting parallelism and elimination of redundant processing become apparent. Thus, our approach can be used to optimize existing implementations by casting the underlying protocols in our framework, and is particularly useful in developing implementations for new protocols.

We believe that our model opens up a new dimension of techniques for high performance protocol implementations without compromising their correctness.

[6] Unlike the existing notion of 'protocol conformance testing' where a protocol implementation is tested for its compliance with the protocol specifications, the problem we address here is how a protocol implementation incorporating performance optimization techniques conforms to the native implementation from which it is derived.

## References


[1] **N. Jain, M. Schwartz, T. R. Bashkow** *Transport protocol processing at Gbps rates* **In proc. ACM SIGCOMM'90. pp. 188-199, 1990.**

[2] **D. Clark, M. Lambert, and L. Zhang.** *NETBLT: A High Throughput Transport Protocol.* **In proc. ACM SIGCOMM'87, pp.353-359, Aug. 1987.**

[3] **J. B. Carter and W. Zwaenepoel.** *Optimistic Implementation of Bulk Data Tranbfer Protocols.* **ACM Sigmetrics Conference, 1989.**

[4] **D. D. Clark and D. L. TennenHouse.** *Architectural Considerations for a New Generation of Protocols.* **In proc. SIGCOMM'90, pp. 200-208, 1990.**

[5] **K. Ravindran, K. Bhat, T. J. Gong, and K. Gould.** *Performance Engineering of End-Systems for High Bandwidth Multimedia Communications,* **Tech. Report, Dept. of Comp. & Info. Sciences, Kansas State University, Oct. 1995.**

[6] **C. M. Woodside, K. Ravindran, R. G. Franks** *The protocol bypass concept for high speed OSI data tranbfer* **Protocols for high speed networking II, IFIP '91. pp. 107-122.**

[7] **C. M. Woodside and Y. M. Thia.** *A Parallel Optimistic Bypass Architecture (POBA) for High-Speed Bulk Data Protocol Processing.* **Tech. Report, Dept. of Syst. & Comp. Eng., Carleton Univ, Nov. 1993.**

[8] **M. Zitterbart.** *High-Speed Protocol Implementations based on Multiprocesor-Architecture.* **Proc. of the IFIP WG 6.1/WG6.4 int. Work. on Protocols for High-Speed Networks, Zurich, Switzerland, May 1989, North-Holland.**

[9] **D. Giarrizzo, M. Kaiserswerth, T. Wicki, and R. C. Williamson.** *High-Speed Parallel Protocol Implementation.* **Proc. of the IFIP WG 6.1/WG6.4 int. Work. on Protocols for High-Speed Networks, Zurich, Switzerland, May 1989, North-Holland.**

[10] **D. Feldmeier.** *A Framework of Architectural Concepts for High-Speed Communication Systems.* **IEEE Journal on Selected Areas in Communications, Vol. 11, No. 4, May 1993. pp. 480-488.**

[11] **K. Ravindran and R. Steinmetz.** *Object-oriented Communication Structures for Multimedia Data Transport.* **IEEE Journal on Selected Areas in Communications, Sept. 1996.**

[12] **S. W. O'Malley and L.L. Peterson.** *A highly layered architecture for high-speed networks.* **Protocols for High-Speed Networks II, M. Johnson, Ed., Palo Alto, CA Nov, 1990. Amsterdam, The Netherlands: North-Holland pp. 141-156.**

[13] **M. Bjorkman and P. Gunningberg.** *Locking Effects in Multiprocessor Implementations of Protocols.* **In proc. ACM SIGCOMM'93, pp.74-83, Sept. 1993.**

[14] **D. Clark, V. Jacobson, J. Romkey, and H. Salwen.** *An Analysis of TCP Processing Overhead.* **In IEEE Comm. Magazine, vol.27, pp.23-29, June 1989.**