Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices

RICHARD W. WATSON
Lawrence Livermore National Laboratory
and
SANDY A. MAMRAK
The Ohio State University

End-to-end transport protocols continue to be an active area of research and development involving (1) design and implementation of special-purpose protocols, and (2) reexamination of the design and implementation of general-purpose protocols. This work is motivated by the perceived low bandwidth and high delay, CPU, memory, and other costs of many current general-purpose transport protocol designs and implementations. This paper examines transport protocol mechanisms and implementation issues and argues that general-purpose transport protocols can be effective in a wide range of distributed applications because (1) many of the mechanisms used in the special-purpose protocols can also be used in general-purpose protocol designs and implementations, (2) special-purpose designs have hidden costs, and (3) very special operating system environments, overall system loads, application response times, and interaction patterns are required before general-purpose protocols are the main system performance bottlenecks.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications, network operating systems*; C.4 [**Performance of Systems**]: *design studies*; D.4.4 [**Operating Systems**]: Communications Management—*message sending, network communication*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Interprocess communication, performance of communication protocols, transport layer protocols—design and implementation

1. INTRODUCTION

Communication between distributed programs (processes) requires mechanisms to transport data end-to-end between source and destination with appropriate error control, resource management, security, and other services. This data may be requests for service from a client process to a server process, a corresponding

This work was performed by Lawrence Livermore National Laboratory under contract number W-7405-Eng-48 under the auspices of the U.S. Department of Energy.

Authors' addresses: R. W. Watson, Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550; S. A. Mamrak, The Ohio State University, Columbus, OH 43210.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1987 ACM 0734-2071/87/0500-0097 $00.75

reply, terminal messages, or a large file. At one end of a spectrum of designs are the clearly identifiable transport protocols operating in a range of network environments and capable of supporting a wide variety of application or interface semantics [3, 13, 14, 33, 36]. At the other end of the spectrum are more specialized transport mechanisms, possibly distributed among many modules in an end node, that are tailored to specific network environments and application or interface semantics [1, 2, 4, 21, 26, 28, 30]. For the purposes of this paper we call the former *general-purpose* (full functionality) transport protocols and the latter *special-purpose* or *problem-oriented* (reduced functionality) transport protocols.

In either case, certain end-to-end issues must be dealt with: (1) deciding what abstractions are being communicated (i.e., messages, packets, byte streams), (2) identifying the communicating parties, (3) detecting and recovering from possible errors, (4) dealing with the management of resources such as buffer space, (5) synchronizing the communicating parties, and (6) protecting the information against unwanted access or modification [32]. One transport service requirement is to move uninterpreted application data units, called messages. Whereas some transport designs include a mechanism to match request and reply messages or related strings of requests and replies, we assume this function to be primarily a higher level protocol issue. We focus here on the functions of connection management, error and flow control, and message segmentation/reassembly because these have been of most concern to date. The mechanisms available for implementing transport functionality are largely independent of the wide range of communication interface primitives and associated semantics that exist to support distributed applications [1, 2, 9, 11, 12, 18, 19, 28, 30, 32].

The work on special-purpose protocols has been motivated by the perception, with which we concur, that many current general-purpose transport protocol designs and/or implementations are not as efficient as desired for many distributed system applications. Even in the areas of file transfer and virtual terminal access, application-specific implementations of general-purpose protocols or application-specific protocol designs have been felt necessary to achieve the desired efficiency [8, 24]. The term efficiency covers a range of characteristics: CPU and packet exchange overhead affecting throughput and delay, code size, and the size of state space.

In general, there are three basic ways to improve efficiency: (1) use improved protocol mechanisms to achieve a given functionality, (2) eliminate unneeded functionality, or (3) use better implementation techniques (these are usually independent of functionality).

Efforts to gain transport service efficiency have used all three approaches. We argue that general-purpose protocol designs and implementations can be made as, or nearly as, efficient as special-purpose ones by new syntheses of improved protocol mechanisms and implementation techniques; that strategies to gain efficiency by reducing functionality result in designs and implementations that have several drawbacks; and that there are few or no effective global system performance gains in a majority of currently existing operating environments.

If our arguments are valid, this question can be asked: Why have researchers in distributed systems, who have judged existing general-purpose protocols unsatisfactory, given more attention to developing special-purpose protocols rather than designing new or improving existing general-purpose ones and their implementations? Further, why are the performance numbers reported in the special-purpose literature usually so much better than general-purpose numbers? One can only speculate on some answers that we believe to be historical, technical, sociological, and organizational.

Historically, most general-purpose protocols have been designed and implemented by people with a communication orientation for relatively slow, higher error rate, wide-area networks, where the dominant applications have been long-lived and session-oriented, virtual terminal, and file transfer connections. The programming interface has been of secondary interest and has usually been based on a device I/O model. This group has focused more on connection-oriented (virtual circuit) protocol architectures, with designs and implementations tuned to this model [38].

Most of the special-purpose protocols, on the other hand, are being developed by people with a language, distributed operating system, or distributed application orientation for high performance local networks, where important applications require short-lived, transaction-oriented request/reply, or page-level file-access style interactions. In the distributed operating system paradigm, terminal access and file transfer are special cases of requests to servers [3, 32]. This group has usually built on a procedure call-like interaction model to which they have tuned their designs and implementations. Remote procedure-call style systems can, of course, also be built on general-purpose protocols [20, 35].

One technical result of these differences is that many general-purpose protocols use more packet exchanges than are felt necessary to reliably exchange a request and a reply. Another technical difference is that the general-purpose implementations are often tuned for bandwidth not latency, as is required for a request/reply style of interaction. Further, most existing general-purpose protocols do not offer the mechanisms that could allow their functionality to be dynamically optimized to the needs of specific application or interface semantics. We argue below that these are not inherent problems with general-purpose protocols.

An important sociological and organizational factor is that existing designs and implementations of general-purpose protocols are not usually under the control of those doing distributed systems research. Often the designs are controlled by a large research community, government standards, or corporate organization; therefore, it is difficult to experiment with them. Similarly, existing general-purpose protocol implementations are often controlled by others, tuned to other applications, difficult to change, and unnecessarily complex and therefore difficult to understand. Even if existing implementations are open to change, such change is viewed as a secondary research interest. It is usually a good research strategy to simplify the factors being studied, and this simplification often leads to the development of special-purpose protocols.

Finally, developers of special-purpose protocols have often had control of the operating system (including device drivers), communication interface, protocol, and even microcode design and implementation. They then compare their performance gains against general-purpose protocols implemented by others that have not been tuned for their application. There is a growing awareness that protocol performance depends as much, and usually more, on the implementation than on the design [6, 16, 24]. Owing to this dependency, it is difficult to evaluate design approaches for lack of well-controlled implementation and environment experiments. For example, some performance numbers reported for special-purpose protocols assume very special case low-level implementations or unique network interface hardware not widely available. They ignore factors common under real operational environments: user-to-system context switches, processor multiplexing for system tasks other than network access, checking of access rights, interrupt handing, limitations of commercially available network interfaces, data copies required in most existing operating environments, and so forth.

Experience with many existing operating systems, designed before the importance of interprocess communication was fully recognized, indicates that the dominant overhead (as high as 80 to 90 percent in our experience) is in the operating system structure, the interface-imposed overhead, and in the lower level network device drivers, rather than in the transport protocol algorithm [6, 7, 16, 24]. These operating systems suffer from poor to nonexistent support for asynchronous I/O system calls, lightweight tasking, memory sharing, interprocess communication, and efficient timer and buffer management. Therefore, there has been little motivation to improve the protocols.

Now that established operating systems must work in local network and distributed system environments, their functionality, implementations, and interface semantics in support of communication are evolving. As a result, there is renewed activity in transport protocol design and implementation, although much of the new work has come from specialized protocol design and implementation. We argue that what is needed is the incorporation of the insights gained from the latter excellent work, as well as experience with existing general-purpose protocol designs and implementations, into more efficient and flexible general-purpose designs and implementations. We would also like to encourage wider publication of protocol implementation and performance experience. Only recently have papers discussing protocol implementations in local network environments begun to appear [2, 3, 5–8, 12, 16, 20, 24, 37, 39].

Our discussion is organized as follows. Section 2 discusses transport service mechanisms with an emphasis on improving general-purpose transport protocol efficiency, thus removing some of the major problems special-purpose protocol designers have found with general-purpose designs. Section 3 describes various implementation strategies for making efficient execution available to both general and special-purpose protocols. Section 4 summarizes our argument. The Appendix contains some measurement data on a particular transport protocol and implementation used to support our analyses.

2. TRANSPORT SERVICE MECHANISMS

Figure 1 summarizes families of mechanisms available for implementing transport protocol functionality. Each is briefly discussed below.

Two costly aspects of packet communication are packet handling and state retention. Packet handling is expensive because of possible user/system context switching, data copying, protocol processing, checksumming, buffer management, and device driver overhead and so forth. State retention is expensive because of

Function	Mechanism
Connection management	Handshake-based Explicit timer-based Implicit timer-based
Errors	Checksums for damage Explicit acks to prevent loss Implicit acks (e.g., application-level replies to requests) to prevent loss Sequence numbers or other identifiers to prevent duplication and missequencing.
Flow-control	Explicit sliding window Implicit (e.g., one outstanding request/reply at a time) Assumptions about relative sender and receiver transmission rates, with discard on overflow Transmission-rate based
Message size	Message and packet segmentation/reassembly Small, fixed, or maximum-size messages and packets

Fig. 1. Mechanisms for implementing transport functionality.

memory requirements (of less importance, given today's large low-cost memories). Control packets (associated with protocol operation) and data packets are both expensive to handle, although the former are less expensive because they may not require user-to-system context switches and data copying or checksumming. Designers of special-purpose protocols have tried to reduce packet exchange overhead by minimizing the number of packet exchanges required for connection management and acknowledgment. Owing to the expense of packet handling, it is widely recognized that to maximize data throughput the largest possible packet sizes should be used. Unfortunately, many implementations of network interfaces restrict packet size (e.g., 1,500 bytes for Ethernet). The reduction of the state required and the interval over which it must be retained has focused on reducing protocol functionality, improving connection management mechanisms, and sharing of connections. These issues are discussed in the following sections.

2.1 Connection Management Mechanisms

Connection management deals with the subtle end-to-end issue of allocating, synchronizing, and deallocating state, primarily identifiers needed for error and flow control [29, 31]. It also involves negotiating modes of operation and needed resources such as packet or buffer sizes [8]. Unless an error and crash-free environment is assumed, connection management issues must be dealt with at some level. The connection management error control problem that must be solved is that receivers maintain state long enough (timer-based) or check with the sender (handshake-based) so that duplicate packets cannot cause duplicate data to be accepted. The ambiguity problem to be solved is assuring that senders maintain state long enough to receive all acknowledgments of data sent, and receivers stay open long enough to receive and acknowledge (ack) all possible retransmissions. This eliminates ambiguity about whether or not data were received, except when an end node crashes or the network is partitioned. Recovery from these problems requires a higher level mechanism [15, 22, 25]. One must also assure that duplicate acknowledgments from old connections cannot acknowledge data sent on a current connection. Reliable connection management, for both general and special-purpose protocols, can be achieved using combinations of message exchange (handshaking), timer, and unique connection identifier mechanisms. Special-purpose protocols eliminate unnecessary connection-management packet exchanges by using assumptions about network error characteristics and a combination of the above mechanisms [1, 2, 26, 30].

2.1.1 *Packet-Exchange Based Connection Management.* Most existing general-purpose transport protocols utilize explicit connection opening and closing packet exchanges [29]. The opening packet exchange guards against opening due to duplicate packets and allows resource negotiation; the closing exchange assures that all data have been received, and both parties are prepared to close. Handshake-based connection management is expensive in terms of packet exchange overhead, delay (depending on assumptions about possible network errors, requests and data may not be reliably delivered until the third packet of the opening exchange), and the extra overhead and implementation complexity required to cycle through the opening and closing handshake states.

If one assumes a general network environment where packets may be duplicated and missequenced, then five packets plus the timer mechanism are required to reliably handle connection management for the exchange of a request and reply [29, 31]. If one assumes no network duplication and missequencing, the number can be reduced to a three packet exchange plus the use of unique connection identifiers and a timer mechanism [1, 2]. When designing special-purpose protocols, one must be explicitly aware that they depend for correct operation on such a mechanism and error assumptions. The combination of unique identifier, handshake, and timer-based approaches commonly used in special-purpose protocols is discussed further in Section 2.2.

However, even when handshake-based connection management is used, careful implementation can minimize the number of exchanges used. For example, while theoretically the Transmission Control Protocol (TCP) can safely exchange a request and reply using five packets, many implementations require nine [13, 29]. The costs of handshake packet exchanges may be acceptable or even desirable in many environments because (1) the patterns of communication may allow these costs to be spread over several data exchanges, (2) the intervals between requests or request and response may be long relative to opening and closing overhead, (3) the system as a whole may be lightly loaded, and (4) the communicating parties can achieve improved overall performance by negotiating transfer parameters and resource allocations.

2.1.2 *Timer-Based Connection Management.* Reliable connection management can be achieved with no extra packet exchanges by using a timer-based mechanism [3, 10]. Timer-based connection management mechanisms have the advantages of requiring no extra delay before data delivery, minimizing packet exchange, small implementation size (see the Appendix), and minimizing state retention when connections are inactive, since state is automatically allocated and deallocated. Timer-based connection management can be used in either special or general-purpose protocols.

In a timer-based connection management approach, the receiver keeps state (e.g., sequence numbers (SNs) or connection or transaction identifiers, or responses) until all old duplicate packets (requests) have died (including retransmissions). The sender keeps state until it can receive an ack if sent (assuming a graceful close is desired), and, depending on the details of the protocol, possibly long enough to guarantee it will generate acceptable SNs or other identifiers. Depending on the protocol design, when a node recovers from a crash it may have to wait a period before sending or receiving to avoid connection management hazards [3, 10, 31]. The main problems that must be solved are determining timer intervals for a given timer-based design and bounding packet lifetime. In an explicit timer-based design, the intervals are explicitly derived and limited on the basis of parameters such as maximum packet lifetime, retransmission time, and acknowledgment time [10]. These intervals are simply bounded [27, 33]. Implicit designs depend on engineering judgments made on bounds for the above intervals [1, 2].

2.1.3 *State Retention.* The amount of state that must be maintained per connection depends on the services being supported (e.g., error control, flow control, encryption, multiple or only single outstanding requests) and protocol-independent implementation choices. As memories increase in size and become cheaper, state retention is becoming less of an issue. Transport level state may be minimized by limiting functionality or by moving functionality to higher levels (in which case the state still exists). It may also be minimized by multiplexing many conversations on one end-to-end transport connection. Performing such multiplexing requires some restrictions such as only allowing one active communicating process per node, allowing only one outstanding request/reply transaction at a time, or a willingness to suffer the delays caused by flow control interaction between the multiple conversations [1, 3].

The length of time state is maintained depends on the connection-management mechanism used. In a handshake-based protocol, state may be discarded after completion of the closing handshake, although some handshake protocols may also require timer-controlled state deallocation to avoid hazards [13, 29, 31]. Use of an implicit or explicit timer-based protocol automatically allocates and deallocates state without the overhead of a connection closing handshake. An explicit timer protocol can reduce the state-time product over an implicit approach since the latter must make worst-case packet lifetime assumptions.

2.2 Error Control Mechanisms

Information may be damaged, lost, duplicated, and missequenced. Designers of special-purpose transport mechanisms or transport protocols with many levels of service try to take advantage of the precise error tolerance of the application and error characteristics of the underlying network or link layer to minimize mechanism. Detecting damaged information and handling lost information are the most expensive services in terms of mechanism or central processing unit time (see Appendix). Detecting and discarding damaged information at any level leads to its loss. Once one provides mechanisms to protect against lost information, duplication is possible due to retransmissions. Duplicate protection during data transfer is, however, quite inexpensive. Once duplicates are protected against, protecting against missequencing is essentially free. In our experience, the basic choice is between assuming an error-free lower level environment or providing the full range of error control services, possibly with limited options. The main design issue is how and at what level to provide these services.

One argument advanced for the use of special-purpose transport mechanisms, implemented within the application, is that no purely transport level protocol can eliminate the need for related higher level services (e.g., in end node or network crash situations). Higher level error recovery mechanisms are required if such situations are to be protected against. Thus, it is argued, if higher level mechanisms are going to be used anyway, why introduce a duplicate mechanism at lower levels [23].

Separate lower level mechanisms cannot be justified in all cases, but the higher level mechanisms may be more expensive. For example, the mechanisms required to protect against information loss in event of node or network crash are quite complex and expensive and must be built into all communicating client/server processes [15, 22, 25]. Many applications do not require such protection. Even when provided, lack of a lower level mechanism may result in more frequent than necessary invocation of the expensive higher level mechanisms, and, therefore, yield potentially higher costs.

There is clearly a trade-off between providing related services at more than one level or only at the highest level. We believe that this trade-off for many existing environments favors the use of lower level mechanisms to improve performance.

2.2.1 *Error Control Mechanisms: Damage.* Damaged messages or packets are normally detected through the use of checksums and are discarded. Recovery is through the use of acknowledgment and retransmission. Most link-level protocols and some local network interfaces support checksums in hardware. Errors can still occur within hardware interfaces or within intermediate nodes in more complex topologies. Thus, end-to-end software checksums are usually provided in both general and special-purpose designs. Software checksums, even those using a simple arithmetic algorithm (see the Appendix), are expensive, a fact that argues for microcode support [28].

Because the undetected link level, interface, and memory error rate of some environments is low enough, end-to-end checksums may not be needed. Optional end-to-end checksums are easily supported for both special and general-purpose protocols [13, 14, 33].

2.2.2 *Error Control Mechanisms: Loss.* The central error control question (once connection management has been dealt with) is: Can packets be lost? If packets can be lost, then a positive-acknowledgment retransmission mechanism must be supported, or the application must be such that occasional lost information is tolerable. Detection and recovery from lost packets is expensive and complicated, requiring acknowledgments (acks), retransmission queues, and timeouts. Further, duplication and missequencing are possible as a result of retransmission.

Two variants of positive-acknowledgment retransmission protocols exist. The most common is to use an ack to acknowledge all data received up to a given sequence number. The loss of one packet can result in retransmission of that packet and all packets sent after it. Another approach is selective acknowledgment or negative acknowledgment (nak), wherein specific packets can be acknowledged or be identified for retransmission. This approach has been widely used on long delay networks such as satellite networks. This approach has been proposed to improve the performance of bulk data transfer for local networks also [3, 8, 37, 39].

We know of no practical proposal for network communication that has successfully assumed no information loss, even for local network environments (local network interfaces can have high losses [16, 39]), unless the expensive loss protection mechanism has just been moved to a higher or lower level. The lost event may be rare, but it occurs. Once its occurrence is assumed, most applications require mechanism be provided to protect against it.

Because packet handling is expensive, one common approach of special-purpose protocols is to try to reduce the number of required acks by using the receipt of an application level reply to provide the ack of the corresponding request. Acks may be saved, but the expensive retry queue and timeout mechanism is also just moved to the higher level.

The problem still remains of acknowledging the response. Because reponses can be lost, the connection-management mechanism is required since duplicate requests will be generated. State must be retained to detect the duplication and retransmit the response, or request duplication must be acceptable (i.e., requests must be idempotent). With assumptions that there is no network duplication or missequencing and that only a single outstanding request is allowed at a time, the connection management mechanism often adopted by special-purpose protocol designers is the following. They use receipt of the next request, if it follows soon enough, to be an implicit ack of the last response, or otherwise use an explicit ack of the response so that the responder can know when to discard state [1, 2, 3, 28]. We expect the latter explicit ack to be required in most situations because it is only for applications with a small expected interval between requests that the former could be used. Since the response-ack can be lost, the implicit or explicit timer mechanism is ultimately required to know when the state can be safely discarded. If responses or acks can be duplicated, then requesters must hold state until they expire or use unique SNs, connection, or transaction identifiers so that the old duplicates cannot ack new requests or data sent.

Another serious loss recovery problem is determining the appropriate timeout for retransmissions, a complicated problem in both wide and local area networks [16, 37]. Ack or reply generation time is more variable if handled at the application level because delays are more dependent on the application, system load, and system resource scheduling characteristics. Thus, it may lead to large topology or application-dependent retry timeout tables, long timeout periods (drastically affecting throughput or delay when even small percentages of packets are lost), or unnecessary retries. Since transport level acks are still required to avoid such problems, most special-purpose protocols contain a rather complicated special-case timeout and explicit ack-generation mechanism [1, 2, 30].

The special assumptions and mechanism used to avoid acks may not improve overall system performance. Depending on the implementation, ack generation and processing may be quite inexpensive. Processing acks will not cause detrimental system delay and CPU overhead if their generation and processing can be overlapped with data copying [39], waiting for responses at the client end, or waiting for I/O at the server end. Acks are not a problem if the time intervals between remote operations or service reponse times are long, relative to the times required to generate and process an ack. Finally, ack expense is less significant if applications run under operating systems in which system call overhead is already unavoidably high. We believe that many distributed applications or environments have one or more of the characteristics above.

In fact, it is precisely in the high-load situation, claimed to benefit from special-purpose protocols, where lack of low-level acks may result in unnecessary retries or another recovery mechanism being invoked, these approaches further end node load and network congestion. For this reason, Casey [2] abandoned use of replies as acks of requests and instead adopted explicit transport level acks of both requests and replies.

One should also recognize that explicit acks have utility. If they are not supported in a special-case protocol, then connection management hazards may exist. They can reduce the time required to hold state, including requests or replies being held for retransmission. They can, as mentioned, reduce retransmissions and their packet handling load and thus reduce congestion during periods of high server load. Once provided, they make support for arbitrary-sized messages and intra and intermessage flow control quite inexpensive. Generating and processing a transport level ack can be less-CPU expensive (by a factor of three in the example in the Appendix) than an ack for requests or replies. This is due primarily to the fact that acks carry no data that must be copied, checksummed, or placed on a retransmission queue. Nor are context switches between user and system space required in kernel implementations.

If minimizing acks is determined to be important, general-purpose transport protocol designs or implementations can do so in at least two ways. One alternative is for the receiver to delay the ack. This increases the probability that the ack can be piggybacked with the response to a request (usually difficult to achieve [6, 7]) or that a single ack can acknowledge several received packets. A delayed ack mechanism can be provided in the implementation with no change to protocol functionality. If the delays to process a request or generate another request are such that a separate ack is required (we expect this to be the normal case in most applications), then the overhead of the additional ack is probably inconsequential, as already mentioned. Delaying the ack can, however, seriously reduce performance if the sender is using smaller buffer space than the receiver and will not release buffers for new data to be sent until it receives the ack.

Another design alternative is to place in the transport level interface send function a boolean parameter indicating whether or not the transport level ack and retransmission mechanism needs to be used for this buffer or message. This flag can then be propagated in the transport protocol header to provide information to the receiving protocol module about whether or not to generate an acknowledgment [36]. The Versatile Message Transaction Protocol (VMTP) provides several control flags to precisely control when acknowledgments should be generated [3]. Such flexibility, however, requires additional higher level mechanism and complexity for effective use.

2.2.3 *Error Control Mechanisms: Duplication.* Duplication results primarily from retransmission at various protocol levels to recover from lost information. Efficient detection of duplicates requires use of an SN either from a contiguous or at least monotonically increasing space. Detecting duplicates requires minimal state and a simple comparison of the received SN against the next expected SN or some acceptable lower bound. Duplicates are discarded, and an acknowledgment (ack or reply) is retransmitted. The main, and often subtle, issues associated with duplicate detection are those of connection management discussed above.

It is sometimes argued that for certain classes of request, called idempotent, duplication causes no difficulty (e.g., reading or overwriting data as opposed to incrementing data) [30]. Unfortunately, many types of request cannot be formulated in this form, or many special cases must be handled. Even when idempotent requests can be appropriately formulated, certain subtle synchronization problems are possible owing to concurrently executing duplicate requests (orphans) [26]. To eliminate these problems one must eliminate duplicates, provide synchronization mechanisms to assure receipt of only the reply to the last request, or provide a mechanism, usually expensive, to kill duplicate executing requests. Whether with respect to the issues of connection management, data transfer, or orphans, no mechanism is saved by performing duplicate detection at a higher level.

2.2.4 *Error Control Mechanisms: Missequencing.* Missequencing is caused by transport level loss and retransmissions if multiple messages can be outstanding, or by lower level network store and forward delays and alternate routing. It is assumed that one of the simplifications introduced by many local network topologies or a single request/reply interaction is that missequencing cannot occur. This assumption has been used to try to simplify connection management [2] and packet acceptance handling. However, the CPU time and mechanism to deal with missequencing, ignoring connection management, is trivial, particularly once packet loss and duplication must be dealt with. We argued above that duplication must be dealt with if the retransmission mechanism is used in end or intermediate nodes. The additional requirement imposed to handle missequencing is the use of a contiguous, rather than simply a monotonically increasing, SN space. This may require additional state space at the sender because a single monotonically increasing state variable cannot be used across all conversations.

2.3 Flow Control Mechanisms

Flow control mechanisms include a priori or negotiated agreements, special messages, or state piggybacked on acks. Since the implementation of buffer and ack generation management interact strongly with flow control, implementation complexity and performance problems are created that special-purpose protocol designers have tried to minimize. An example given earlier was the interaction of sender and receiver buffer sizes, window flow control, and use of delayed acks. Another example is the *silly window syndrome* in which the interaction of buffer, window, and ack strategies can result in very small packets being sent [5]. Simple changes to an implementation can solve both problems [3, 5].

Four restrictions and corresponding optimizations have been used in various combinations to simplify the interactions of the above factors with flow control and to minimize state and acks: (1) a priori or negotiated agreement on the maximum size of buffers, packets, or request or reply messages transmitted (e.g., assume the receiver can buffer all packets or messages received up to a given amount of data), (2) specify a priori agreement on the number of messages allowed to be outstanding at a time (e.g., most commonly use a stop-and-wait protocol that only allows one message or packet to be outstanding at a time), (3) use a blast protocol that allows one large message or buffer, sent as many packets, to be outstanding and generates one ack at the end of this message or buffer (it may use rate-based flow control to control the intramessage packet arrival rate; the appropriate rate can be negotiated between sender and receiver or be adaptively determined [3, 8]), and (4) use no flow control and simply discard a packet buffer or message if a receiver has not allocated space for it, and depend on loss recovery to handle the problem.

Discard seems particularly unacceptable because it is precisely in the heavy load situations assumed favorable to special-purpose protocols that flow control is needed to reduce congestion. Discard will lead the requesting end to invoke expensive loss recovery, for example, and to poll by retransmitting until an ack or reply is received, thus increasing packet handling overhead and congestion.

If size restrictions are too small in cases 1 and 2, data throughput is limited. It has been argued that in a low-delay, high-bandwidth local network a stop-and-wait protocol can achieve throughput competitive with sliding window or blast protocols [1]. This is not the case, as shown by Zwaenepoel's studies in which sliding window and blast protocols achieve significantly better throughput in a local network than stop-and-wait protocols (sliding window and blast protocols achieve roughly equivalent results) [39]. Sliding window and blast protocols achieved their advantage by being able to make use of data pipelining and concurrent sending and receiving at each end. If a stop-and-wait protocol is used for requests/replies, then a separate transport mechanism may be required to achieve acceptable bulk data transfer rates [4]. This use of multiple protocols leads, in our view, to unnecessary, extra conceptual and implementation complexity.

One of the arguments for use of the stop-and-wait mechanism and assumptions on maximum packet or message size is that they help eliminate the need for extra flow control messages or explicit acks. We argued earlier that eliminating acks may be detrimental under high loads and provide little overall system gain under low loads. One view of assumptions on the use of large buffers, in case 1, and the use of blast protocol mechanism, in case 3, is that this is just a special case of a sliding window protocol with a large default window and delayed ack generation. The stop-and-wait mechanism can also be viewed as a special case of a sliding window mechanism with a window for one packet or message.

Thus, performance problems with sliding window mechanisms do not seem to be inherent; that is, they should be able to achieve good performance for a variety of applications. Instead, the problem seems to be in getting the implementation correct so as to minimize the negative interaction of buffer and ack management with advertised windows [5, 8]. Achieving such an implementation would seem to require hints at the user interface as to the nature of the application's message size and interaction characteristics [16]. Further, to achieve good performance, both ends need to get it right. Implementation experience to date indicates that this is a difficult task. Much of the problem seems to result from a difficulty in synchronizing or coordinating, in both design and implementation, the interactions of the abstractions used at the different levels; the application deals with application-level buffers or messages, most lower level mechanisms deal with their own buffers and buffer management strategies, and sliding-window flow-control protocols specify flow control on yet different abstractions such as bits, bytes, or packets.

Our conclusion is that some explicit form of flow control is desirable especially under high loads. Because of the subtle interactions of flow control, buffer, ack generation, and other mechanisms, we feel flow control and its implementation is still an area not adequately understood and in need of new ideas and more modeling and implementation experimentation. The blast mechanism appears promising for both general and special-purpose use, given the ability to assume large default send and receive buffers, although most of its basic concepts can be included in sliding-window protocol implementations with appropriate ack and nak mechanism and management.

2.4 Message Size

If messages can be restricted in length and fit in a single packet, then no mechanism is required to handle the error and flow control problems of intramessage segmentation. The transport mechanisms can then deal with the message, rather than the packet, as the abstraction to be transported, and the protocol does not need mechanism for error and flow control for pieces of a message or to support a message segmentation/reassembly mechanism.

It is our experience (see Appendix) that the overall cost of dealing with message segmentation/reassembly and intramessage error and flow control is not significant. Assuming that all messages are less than some fixed packet size (often 512 bytes or less) places restrictions on higher level application and service design or implementation. It may just move segmentation/reassembly to a higher level. It also implies that bulk data transfers must be handled using many pairs of requests and replies, reducing performance, or requiring a separate mechanism. Provision of support for arbitrary message sizes and intramessage flow control (either sliding windows or rate based) leads to more efficient data movement [3, 39].

3. IMPLEMENTATION STRATEGIES FOR GAINING EFFICIENT EXECUTION

Good performance requires careful implementations of both general-purpose or special-purpose protocols. In fact, the implementation seems more important than the design. (We are familiar with implementations of the TCP [13] and

ISO transport protocols [14] in which implementation size and throughput can vary by an order of magnitude for the same protocol implemented by different groups for the same operating system.) Several areas of implementation optimization have been suggested or tried and are reviewed below. The same implementation optimization techniques are available to both classes of protocol implementors.

3.1 Layered Architectures

Design decisions on modularization and placement of protocol processing can significantly affect performance. General-purpose layered protocol architectures have often been implemented with separate (often user-level) processes per protocol layer or connection. This can lead to inefficient operation. Such a modularization is an implementation design choice and is not inherent in general-purpose protocols. That is, a common mistake is to take a layered design as a requirement for a correspondingly layered implementation.

One of the claims often made for special-purpose protocols is that they are not layered and therefore are more efficient. In fact, examination of their designs shows that they are cleanly layered, at least at a logical level of abstraction sense. What they seem to be saying is that, whereas they are designed in a functional sense in a modular layered fashion, these protocols are not necessarily implemented on layer boundaries. It seems important to us to make a sharp distinction between design and implementation [6, 7, 33]. It is important to understand well-defined layers of abstraction. If it is useful to combine functions of two or more layers into one or more asynchronous tasks to achieve efficient operation, then that can be done without loss of the ability to multiplex two or more layer $n + 1$ protocols on a layer n protocol.

3.2 Use of Microcode

A related argument is that special-purpose protocols are simple enough to permit their implementations to be pushed into the operating system kernel or even to be placed in microcode for further efficiency [28]. This is true of general-purpose protocols as well, unless their functional design dictates such a large implementation that kernel or microcode memory space limitations preclude such optimization. As shown in [3] and in the Appendix, it is possible to design general-purpose transport protocols that can be implemented within small memory space requirements.

3.3 Lightweight Processes and Context Switching

An expensive operation in remote communications is context switching from the user to the system environment and the software checking that is often required when that boundary is crossed; this operation thus argues for kernel implementations. Even context switching within the system kernel can be expensive, depending on machine and operating system architecture. Arguments are made that the provision of lightweight processes or tasks, which can be efficiently created, destroyed, scheduled, and context switched, support the use of special-purpose protocols. Our experience and that of others shows that, because of the high level of concurrency involved in communication, explicit support for lightweight tasking in the kernel is generally essential to protocol implementations [4, 7, 24]. Lightweight tasking can and should be provided within user-level processes as well, if protocols are implemented at that level. Such mechanism aids both performance and understanding through a cleaner implementation structure.

3.4 Data Copying and Buffer Management

Data copying is expensive and often found in several places such as user-to-system, protocol-module-to-protocol-module, and system-to-network interfaces. Many protocol implementations have suffered because the implementation did not minimize data copying. The ideal is to support data transfer directly from user memory space to the network hardware and vice versa [39]. This may not be possible because (1) the machine's I/O architecture or the network interface adapter's architecture would not allow this, for example, because data chaining was not supported, and thus data and protocol headers could not be appropriately separated or combined; (2) the host operating system's protection, virtual memory, buffering, or other structures required copying; and (3) the protocol implementation modularization required multiple copies.

The protocol implementer has control over (3) and often (2) but not usually (1). We hope that protocol implementation experience can be fed back into the architecture and implementations of (1) and (2) so that they no longer dictate unnecessary copies [17].

As mentioned in Section 2.3, flow control, buffer and ack management, and other mechanisms strongly interact. Choice of a buffer management strategy is an area where implementer control exists. Saltzer et al. discuss how their choices facilitated efficiency gains for a TCP implementation for terminal access and for a simple file transfer from a personal computer [24]. Lantz discusses buffer management strategies for a TCP [16]. Miller and Souza implemented a remote procedure call interface twice, once with a special-purpose transport protocol and, again, on top of a TCP. They indicate a slight preference for the latter approach in their environment, particularly if they tune the TCP buffer management implementation and other areas [20]. Zwaenepoel demonstrates again the utility of double buffering [39].

Since the best buffer management strategy may be application dependent, optimum performance may be facilitated by allowing hints at the user interface [16]. This method may be seen as extra implementation or user interface complexity, but it would seem less complex than requiring separate request/reply and bulk data transfer protocols that are needed by some special-purpose systems.

3.5 Caching

Table lookup to perform network level routing or find connection records and other state is another source of overhead. Often state may be kept in user space, and a context switch may be required. Additional efficiency can be gained if the communication user interface and protocol implementation support caching of frequently accessed state.

4. SUMMARY

General-purpose transport protocols have the advantages that they are portable across environments, can be used with a wide spectrum of applications, can support environmental evolution, and only one protocol suite need be maintained for a broad range of network and application environments. We have seen no evidence that they necessarily imply larger implementations than is possible with special-purpose protocols. Their main disadvantage is that their performance can be expected to be somewhat less efficient than that of special-purpose protocols. There is also a potential increased complexity in the area of flow control that may make it more difficult to obtain correct and optimum implementations. Their complexity in the areas of connection management and error control may be less because their correctness depends on general principles rather than on many special-case environment assumptions. Their performance penalty is mainly a function of which protocol mechanisms are used and of how they are implemented. We argued that a variety of design mechanisms and implementation optimization strategies exist that are applicable to both general and special-purpose designs. These are summarized below.

The primary advantage of special-purpose protocols is their potential for more efficient execution and low delay. The efficiency advantages, from a system point of view, are only realizable in restricted operational environments. The characteristics of these environments, along with a brief explanation of why the characteristic is required, are summarized in Figure 2. They are primarily required to realize savings from special-purpose connection management and ack reduction mechanisms.

Special-purpose designs have the disadvantage that they are closely tied to particular network or application architectures and create difficulties when portability across environments or environmental evolution is desired. We consider this an important disadvantage because our experience is that the underlying network topology and environment is constantly evolving because of the rapid change in technology, constant pressure to create wider interconnection, and growing application needs. Special-purpose protocols also have the disadvantages of potentially introducing complexity due to special-case mechanisms or of failing to guard adequately against all error cases, particularly those associated with connection management.

The design of a general-purpose protocol and its implementation will determine the range of its applicability. Unfortunately, we agree that many existing general-purpose transport protocol designs and implementations do not support the performance desirable and potentially possible for many applications. This fact has been the motivation for the excellent work on special-purpose transport mechanisms and implementations.

We can summarize our arguments for why we believe general-purpose protocols can be designed and implemented to be competitive with special-purpose ones.

Minimizing connection management packet exchanges.

Special-purpose protocols tend to use implicit timer-based connection management to eliminate or minimize the need for explicit packet exchanges. Explicit timer-based connection management and bounding of packet lifetimes achieves the same goal for general-purpose protocols with little additional cost.

Environment Characteristic	Why Required
Static application or network environment	Owing to special case assumptions, there may be performance or correct operation penalties for change.
Primarily homogeneous	Complexity of customization grows quickly as more special cases must be introduced.
High CPU load, short request queues	Otherwise, there is sufficient time to the overlap generation and processing of acks if spare CPU cycles exist (long queues (high load) and lack of explicit acks can compound congestion problems inherent in this case).
Frequent remote requests	Ack overhead is inconsequential for infrequent requests.
Very low service response times	Unnecessary retransmission of requests is very expensive; long service times imply acks are needed anyway and overhead is inconsequential.
Very low service request intervals	Low service request intervals are necessary to enable requests to ack responses, otherwise explicit response-acks are probably required.

Fig. 2. Optimal operating environment for special-purpose protocols.

Minimizing acks.

Special-purpose protocols minimize acks in two main ways: (1) by using application-level requests and replies to ack each other instead of explicit transport-level acks, or (2) by using a single transport-level ack to acknowledge several packets. General-purpose protocols can support interface control over ack generation to achieve the first solution, or can implement appropriate ack management to achieve the second. Our analysis, however, questions the need for the former because in most environments ack generation and processing is less expensive than data transfer and can be overlapped with data copying, waiting for I/O, or waiting for a response, and because, in high load environments, lack of explicit transport-level acks will likely lead to more packet exchanges and corresponding congestion (see Figure 2).

Assumptions about network error characteristics.

Special-purpose protocols simplify mechanism based on assumptions that networks may not damage, lose, duplicate, or missequence packets. Checksumming to protect against damage is expensive and is found in both classes of protocols, but can inexpensively be made optional in either class. Local area networks have good packet loss characteristics, but often poor interface-loss characteristics under heavy load if buffers are overrun [16, 39]. Therefore, both special and general-purpose protocols need to protect against loss for most applications. This need leads to equally expensive loss detection and recovery mechanism and potential duplication, and thus requires a connection management mechanism and sequence numbers or other identifiers. Handling duplicates and missequencing are a low-cost comparison operation.

Flow control and maximum packet or message size.

Special-purpose protocols tend to combine flow control and a maximum packet of message-size restrictions. They either offer no flow control, depending on discard and recovery by retransmission, or only allow a single outstanding packet or request at a time. The former creates severe problems under a heavy load assumed favorable to special-case protocols, while the latter limits applicability or performance, or requires segmentation/reassembly at a higher level. Segmentation/reassembly is not expensive. Improved general approaches are, however, required for flow control and buffer and ack management. This currently means either more carefully specifying how sliding-window flow control, buffer and ack management, and other protocol mechanisms are to be designed, implemented, and used [3, 5, 16] or using a priori assumptions on the availability of large receiver and sender buffers in conjunction with a blast protocol mechanism [3, 8, 37, 39]. Many of the insights gained in the latter can be applied to the former. The area of flow control and buffer management, however, still seems in need of new ideas and additional analysis, modeling, and implementation experience.

Implementation optimization.

Special-purpose protocol implementers have used a variety of mechanisms, outlined in Section 3, to achieve efficient implementation. These mechanisms are also available to general-purpose protocol implementers.

These observations lead us to the conclusion that the choice of a special or general-purpose protocol should be based on the assumptions that can be made about the environment in which the protocols must operate. If it meets the conditions of Figure 2 (expected to be rare), then special-purpose protocols should be seriously considered. Otherwise, it may be best to adapt or use a general-purpose transport protocol. We hope more data will be published characterizing applications and system loads commonly experienced. We also hope that the experience gained with special-purpose protocols and through measurement and analysis of existing general-purpose protocol implementations will be fed back into new, improved generations of general-purpose transport protocol designs and implementations.

APPENDIX

In order to investigate the code size and CPU cycle cost of implementing a general-purpose transport layer protocol, a detailed instrumentation was made of an implementation of a full-service timer-based transport protocol, Delta-*t*, [10, 34] on a DEC VAX 11/750. The statistics below are based on an early implementation in BLISS under VMS 2.5 for handling 1,024 byte data packets. While some tuning was performed, no attempt was made to improve performance by reducing use of procedures, since this would affect modularity. A procedure call and return in BLISS takes about 20–50 microseconds (variable, depending on number of parameters). The instrumentation allowed for isolation and measurement of individual code segments.

The data were used to support several observations made in the main body of the paper. These observations include the following:

(1) general-purpose transport protocol implementations can be relatively small (2,800 bytes for the transport protocol);

(2) acks are relatively expensive, but less expensive than data sending or receiving (20–30 percent of the cost to send or receive a message);

Table I. Code and Data Sizes

Code	Data Sizes
Delta-t (transport protocol) module	2,800 bytes
Nontransport protocol code (includes user interface, packet buffer management, network protocol modules, and a link-level protocol interface, plus 500 bytes to bypass Delta-t when both origin and destination are on the same machine).	3,650 bytes; a link-level protocol module would add 1,500 bytes.
Connection record size	100 bytes per association
VMS unit control block	150 bytes per association
Packet receive buffers	8K fixed
Packet send buffers	Dynamically allocated

Table II. Summary of Times for Sending and Receiving Data (1,024 bytes) and Ack Packets (in Microseconds) on Delta-t

Send a data packet		Send an ack packet	
u/s =	1,167	u/s =	0
s/t =	1,133	s/t =	150
Δt =	565	Δt =	189
n/l =	1,326	n/l =	842
t =	4,191	t =	1,181
Receive a data packet		Receive an ack packet	
u/s =	1,167	u/s =	0
s/t =	1,102	s/t =	468
Δt =	346	Δt =	543
n/l =	1,578	n/l =	1,114
t =	4,193	t =	2,125

Legend: User/system interface time—u/s
System/transport interface time—s/t
Delta-t transport protocol time—Δt
Network and lower level protocol time—n/l
Total time—t

(3) context-switches and system-call software checks are relatively expensive (about 30 percent of the total cost of either sending or receiving a message);

(4) most transport level services are relatively inexpensive; the transport-protocol execution time itself comprises a small proportion of the total time needed to send and receive packets;

(5) since each protocol service was organized in one or more procedures, the procedure call/return overhead is significant.

The user/system interface time is the time for a context switch from user to system and vice versa and for performing checks by VMS. It is a host-architecture dependent time.

The system/transport interface supports the largely protocol-independent send and receive buffer queues, packet buffer and connection-record space management, and copying of data from user space to packet buffers or vice versa.

The transport protocol algorithm is isolated in the Delta-t procedure and could be replaced with minor changes elsewhere by another transport protocol.

Table III. Detailed Measurement Data for Sending a Data Packet

System Component	Action	Execution Time (μs)	
User/system interface (give system an empty buffer)	context switch and VMS QIO systems call overhead	837	
	schedule user software interrupt when send completes	330	
			1,167
System/transport interface (send queue management)	compute next message size	29	
	get connection record	124	
	get empty packet buffer	127	
	call to Delta-*t* to send packet	42	
	move 1,024 bytes to packet buffer	615	
	update send queue data structure	196	
			1,133
Delta-*t* (transport protocol)	decide if and how much data to send (e.g., flow control)	113	
	send timer management	89	
	form data packet header	123	
	set up retry record (loss protection)	240	
			565
Network and lower levels (packet arrives)	adjust packet lifetime	66	
	data checksum (damage protection)	484	
	header checksum (damage protection)	66	
	VAX machine-dependent byte adjustment (due to byte ordering in VAX words)	160	
	routing and lower level functions (estimated)	550	
			1,326
		Total	4,191

Delta-*t* executes mechanisms to handle loss, duplication, and missequencing and supports both arbitrary message sizes and flow control.

The network and lower layers support damage detection (a Ones complement checksum) of packet headers and data and network routing. The time includes an estimate of link-level protocol and channel-driver time for use of an Ethernet.

Network and lower level protocol time may vary widely, depending upon the actual networking hardware. The network/lower level protocol time for sending a packet on a NSC HYPERchannel, for example, is 9.1 milliseconds in one implementation on a VAX 750 running UNIX,[1] which is seven times our Ethernet estimate. The latter is based on our interpretation of some informal measurement notes from the University of California B.s.d. 4.2 UNIX project. The network/lower level time to receive an ack on a HYPERchannel is 3.8 milliseconds or more than three times our estimated value.

To get some idea of what might be gained if arbitrary length messages, flow control, damaged, lost, duplicate, or missequenced packet services were not supported or were made optional, various labeled times could be subtracted. One can see that most of these services are relatively quite inexpensive, particularly once packet loss or damage protection is required. We believe a more efficient implementation of the retransmission queue mechanism is possible.

[1] UNIX is a trademark of AT&T Bell Laboratories.

Table IV. Detailed Measurement Data for Receiving a Data Packet

System Component	Action	Execution Time (μs)	
User/system interface (give system an empty buffer)	user context switch plus VMS QIO system call overhead	837	
	schedule user software interrupt when data received	330	
			1,167
System/transport interface	wake up destination process	135	
	verify connection record has not timed out	123	
	update receive queue state	106	
	move 1,024 bytes to user buffer	512	
	decide if buffer is finished	68	
	release packet buffer	158	
			1,102
Delta-*t* (transport protocol)	determine packet type	14	
	check packet lifetime	41	
	duplicate detection	106	
	flow control and missequence acceptance	100	
	receive timer management	85	
			346
Network and lower levels (packet arrives)	routing and other lower level functions (estimated)	550	
	set packet death time	68	
	machine-dependent byte adjustment processing	136	
	header checksum (damage protection)	56	
	data checksum (damage protection)	457	
	get connection record	149	
	setup and queue software interrupt to Δt	162	
			1,578
		Total	4,193

Table V. Detailed Measurement Data for Sending an Ack Packet

System Component	Action	Execution Time (μs)	
System/transport interface (give system an empty buffer)	decision to ack or not	30	
	get an ack packet buffer	120	
			150
Delta-*t* (transport protocol)	formation of ack	189	
			189
Network and lower levels (packet arrives)	adjust packet lifetime	66	
	header checksum	66	
	machine-dependent byte adjustment processing	160	
	routing and other lower level functions (estimated)	550	
			842
		Total	1,181

Table I summarizes the code and data spaces sizes in the implementation. Table II summarizes the execution time measurements. Tables III–VI present more detailed measurement data upon which Table I is based. The data are organized by logical layer, not by sequence of execution. The legend was chosen

Table VI. Detailed Measurement Data for Receiving an Ack Packet

System Component	Action	Execution Time (μs)	
System/transport interface	notify receiving process	135	
	verify connection record still exists	123	
	send queue management	52	
	release packet buffer	158	
			468
Delta-*t* (transport protocol)	determine type of packet	14	
	check lifetime	41	
	packet acceptance and miscellaneous	111	
	delete retry records (loss protection)	276	
	flow control processing	101	
			543
Network and lower levels (packet arrives)	routing and other lower level functions (estimated)	550	
	set packet death time	68	
	machine-dependent byte adjustment processing	136	
	header checksum	56	
	get connection record	149	
	setup and enqueue software interrupt	155	
			1,114
		Total	2,125

to separate and emphasize the costs that are independent of a specific transport protocol.

ACKNOWLEDGMENT

Alex Phillips of Lawrence Livermore National Laboratories implemented and instrumented the transport protocol from which we obtained the measurement data presented in the Appendix.

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government thereof, and shall not be used for advertising or product endorsement purposes.

REFERENCES

1. BIRRELL, A. D., AND NELSON, B. J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), 39–59.
2. CASEY, L. Remote rendezvous. Tech. Rep., Bell Northern Research, Ottawa, 1985.
3. CHERITON, D. R. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of the SIGCOMM '86 Symposium on Communications Architectures and Protocols* (Stowe, Vt., Aug. 5–7). ACM, New York, 1986, pp. 406–415.

4. CHERITON, D. R., AND ZWAENEPOEL, W. The distributed V kernel and its performance for diskless workstations. *Oper. Syst. Rev. 17*, 5 (Oct. 1983), 128–139.
5. CLARK, D. D. Window and acknowledgment strategy in TCP. Internet Protocol Implementation Guide, Network Information Center, SRI International, Menlo Park, Calif. (Aug. 1982).
6. CLARK, D. D. Modularity and efficiency in protocol implementation. Internet Protocol Implementation Guide, Network Information Center, SRI International, Menlo Park, Calif. (Aug. 1982).
7. CLARK, D. D. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash. Dec. 1–4) ACM, New York, 1985, pp. 171–180.
8. CLARK, D. D., LAMBERT, M. L., AND ZHANG, L. NETBLT: A bulk data transfer protocol. DARPA Network Working Group Request for Comments 969, Network Information Center, SRI International, Menlo Park, Calif. (Dec. 1985).
9. FELDMAN, J. A. High level programming for distributed computing. *Commun. ACM 22*, 1 (June 1979), 253–268.
10. FLETCHER, J. G., AND WATSON, R. W. Mechanisms for a reliable timer-based protocol. In *Computer Networks 2*, North-Holland, Amsterdam, The Netherlands, 1978, pp. 271–290.
11. GENTLEMAN, W. M. Message passing between sequential processes: The reply primitive and the administrator concept. *Softw. Pract. Exper. 11* (1981), 435–466.
12. HOARE, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.
13. INFORMATION SCIENCES INSTITUTE. DOD Standard Transmission Control Protocol. Information Sciences Institute, Marina del Ray, Calif. (Jan. 1980).
14. INTERNATIONAL STANDARDS ORGANIZATION. Information processing systems—open systems interconnection—transport protocol specification. International Standards Organization, ISO/DIS 8073, Rev., ISO/TC 97/SC 16WG 6, June 29, 1984.
15. LAMPSON, B. Atomic transactions. In *Distributed Systems: Architecture and Implementation*, Chap. 11, Springer-Verlag, New York, 1981.
16. LANTZ, K. A., NOWICKI, W. I., AND THEIMER, M. M. An empirical study of distributed application performance. *IEEE Trans. Softw. Eng. SE-11*, 10 (Oct. 1985), 1162–1173.
17. LEACH, P. J. ET AL. The architecture of an integrated local network. *IEEE J. Select. Areas Comm. SAC-1*, 5 (Nov. 1983), 842–837.
18. LISKOV, B. Primitives for distributed computing. *Oper. Syst. Rev. 13*, 5 (Dec. 1979), 33–42.
19. MANNING, E., LIVESEY, N. J., AND TOKUDA, H. Interprocess communication in distributed systems: One view. In *Proceedings of IFIP 80.*
20. MILLER, S., AND SOUZA, R. UNIX and remote procedure calls: A peaceful coexistence? M.I.T. Project Athena Tech. Rep., Massachusetts Institute of Technology, Cambridge, Mass. 1985.
21. POPEK, G., ET AL. LOCUS, A network transparent high reliability distributed system. *Oper. Syst. Rev. 15*, 5 (1981), 169–177.
22. REED, D. P. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst. 1*, 1 (Feb. 1983), 3–23.
23. SALTZER, J. H., READ, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Trans. Comput. Syst. 2*, 4 (Nov. 1984), 277–288.
24. SALTZER, J. H., ET AL. The desktop computer as a network participant. *IEEE J. Select. Areas Comm. SAC-3*, 3 (May 1985), 468–477.
25. SHRIVASTAVA, S. K. Structuring distributed systems for recoverability and crash resistance. *IEEE Trans. Softw. Eng. SE-7*, 4 (July 1981), 436–447.
26. SHRIVASTAVA, S. K., AND PANZIERI, F. The design of a reliable remote procedure call mechanism. *IEEE Trans. Comput. C-31*, 7 (July 1982), 692–697.
27. SLOAN, L. Mechanisms that enforce bounds on packet lifetimes. *ACM Trans. Comput. Syst. 1*, 4 (Nov. 1983), 311–330.
28. SPECTOR, A. Z. Performing remote operations efficiently in a local computer network. *Commun. ACM 25*, 4 (Apr. 1982), 246–259.
29. SUNSHINE, C. A., AND DALAL, K. K. Connection management in transport protocols. *Comput. Networks 2*, 4/5 (Sept./Oct. 1978).
30. WALKER, B. Issues of network transparency and file replication in the distributed file system component of LOCUS. Ph.D. dissertation, Dept. of Computer Science, Univ. of California, Los Angeles, (1983).

31. WATSON, R. W. Timer-based mechanisms in reliable transport protocol connection management. In *Computer Networks 5*, North-Holland, Amsterdam, The Netherlands 1981, pp. 47–56.
32. WATSON, R. W. IPC interface and end-to-end protocols. In *Distributed Systems Architecture and Implementation.* Springer-Verlag, New York, 1981, 140–174.
33. WATSON, R. W. Delta-t protocol specification. UCID-19293, Lawrence Livermore Laboratory, Livermore, Calif., (Apr. 1983).
34. WATSON, R. W., AND FLETCHER, J. G. An architecture for support of network operating system services. In *Computer Networks 4*, North-Holland, Amsterdam, The Netherlands 1980, pp. 33–49.
35. XEROX. Courier: The remote procedure call protocol. Xerox System Integration Standard, Xerox Corporation, Stamford, Conn., (Dec. 1981).
36. XEROX. Internet transport protocols. Xerox System Integration Standard, XSIS028112, Xerox Corporation, Stamford, Conn., (Dec. 1981).
37. ZHANG, L. Why TCP timers don't work well. In *Proceedings of the SIGCOMM '86 Symposium on Communications Architectures and Protocols* (Stowe, Vt., Aug. 5–7). ACM, New York, 1986 pp. 397–405.
38. ZIMMERMAN, H. OSI reference model—The ISO model of architecture for open systems interconnection. *IEEE Trans. Commun. COM-28*, 4 (Apr. 1980), 425–432.
39. ZWAENEPOEL, W. Protocols for large data transfers over local networks. In *Proceedings of the 9th Data Communications Symposium* (Whistler Mountain, British Columbia, Canada, Sept. 10–13). ACM, New York, 1985, pp. 22–32.

Received November 1984; revised November 1986; accepted November 1986