Parallel and Configurable Protocols: Experiences with a Prototype and an Architectural Framework

Bert Lindgren *Bobby Krupczak*
Mostafa Ammar *Karsten Schwan*

GIT–CC–93/22

March 1993

Abstract

As network speeds increase, a major concern regarding communication protocols is their limited throughput and latency due to processing overheads at network nodes. Furthermore, novel network applications and expanded network usage are leading to increased network requirements, particularly with respect to security and bandwidth. In this paper we consider the use of parallelism and configurability to increase throughput and reduce protocol processing latencies. We obtain experimental results of parallel protocol performance using a prototype implemented on a shared memory multiprocessor. The results demonstrate the utility of parallel protocol processing, and they indicate the further research necessary for constructing viable communication protocols for large-scale parallel machines. Based on these experiences, we present the design of an object-oriented framework for parallel protocol programming. This framework both facilitates parallel protocol development and helps maximize protocol performance on a wide variety of multiprocessors.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

1 Introduction

Recent advances in data transmission technology have yielded media bandwidth in the gigabit/second range. However, there has been a lack of similar progress in the area of software technologies for effectively servicing these large bandwidths. As a result, current research is attempting to remove what is now one bottleneck in high-performance communication: the overheads of protocol processing at intermediate and/or end points of communication links.

Another effect of the increasing transmission capabilities and ubiquity of networks is the development of applications that make increasing use of data manipulation such as encryption for security and/or compression for image transmission. It has been argued that such data manipulation can become the bottleneck in protocol processing[Clark-89, Clark-90].

The relatively slow processing speeds of the protocols in predominant use today are largely attributable to the following general features:

- Protocols are designed to process data serially, one packet at a time and one layer at time.
- Protocols are using additional computation in order to reduce the amounts of transferred data.
- Protocols may suffer from excessive functionality, which can lead to substantial increases in protocol execution times due to the need to check for rarely occurring special cases.

Our work attempts to increase protocol processing speeds in three ways:

1. By exploiting the parallelism inherent in protocol processing activities.
2. By defining protocols such that they can be tailored to suit particular application requirements, where such configuration may be performed on-line, off-line, or both.
3. By using non-traditional protocol architectures in order to match protocol configuration and parallelization with application parallelism and with protocol functionality requirements.

The novel attributes of our research compared to previous work on configurable communication protocols are (1) the systematic investigation of parallelism in protocol processing and (2) in relating such parallelization to parallelism in the application, on large-scale parallel machines. Specifically, in this paper we develop a framework with which protocol programmers can create configurable protocols that use *scalable* parallelism to achieve high-speed protocol processing. This framework is developed in response to experimentation with a prototype parallel communication protocol developed on a 32-node BBN Butterfly multiprocessor.

The remainder of this paper is organized as follows. Section 2 describes the basic assumptions of our research and describes some related research, as well as the sample distributed application guiding our research. In Section 3, we present the parallel communication protocol developed on the BBN Butterfly, evaluate its performance, and draw several conclusions regarding the basic issues to be addressed for any parallel communication protocol. In Section 4, we address those issues by presenting an object-oriented framework for the implementation of communication protocols on machines ranging from uniprocessors to large-scale shared memory multiprocessors. This framework is currently being developed by our group on a 32-node Kendall Square Research (KSR) supercomputer, and it is intended to be used for protocol developments targeting future high-performance communication devices attached to this and other machines. Finally, Section 6 of this paper identifies some open issues and comments on the status and future work of our research.

2 Background of Research

2.1 Related Work

Researchers have used several approaches to address the needs of future communication networks, including:

- Protocol adjustments;
- Reference model changes; and
- Implementation improvements.

Protocol adjustments include lightweight protocols that streamline the code that is executed when packets arrive correctly (see [Doeri-90] for a good survey), custom protocols written to support *a specific application*, and *application level framing* which gives the application access to misordered data, so that the application can process it immediately and, therefore, avoid falling behind[Clark-90].

Reference model changes involve modifications to the framework under which protocols are designed. Typically, this involves violating the layered model. For example, in the x-Kernel, a protocol is decomposed into "micro-protocols", which can be combined in order to configure the protocol for each application's needs [Peter-90, Hutch-89b]. Furthermore, Clark and Tennenhouse's "integrated layer processing" suggests that protocols be designed in layers, but then permit protocol programmers to merge such layers during implementation[Clark-90]. Haas' HOPS protocol and Zitterbart's Transputers are also functionally-decomposed protocols, but their functions can be performed in parallel[Haas-91, Zitte-91].

Implementation improvements range from improving the operating system support for protocols [Hutch-89a, Dinge-90, Watso-87] to using parallelism to pipeline STREAMS layers [Preso-92, Garg-90].[1]

Our research builds on the work mentioned above, but differs in several respects. First, we do not limit ourselves to the pipelined parallelism of the STREAMS protocol, or to the functionally decomposed parallelism in the work of Haas or Zitterbart. Such limitations would not be appropriate for large-scale parallel machines. Second, our modifications to the framework for protocol design explicitly address parallelism, in contrast to the x-Kernel. Third, we consider issues regarding data decomposition and movement caused by the use of parallelism, in addition to the problems with reliability and access to misordered data considered by Clark.

2.2 A Sample Distributed Application

We evaluate communication protocols for future networks and network applications in the context of a realistic, large-scale application program. This program executes on a heterogeneous set of distributed supercomputers consisting of multiple parallel and sequential machines linked via high-performance networks. Furthermore, since we are concerned with applications that interact with their human users or developers, this distributed program offers interfaces that execute simultaneously with the program's computational and storage tasks. The computations performed by this application simulate the dynamics of interacting particles. Related applications being developed by other research groups in the U.S. include interactive weather prediction simulations and other engineering or scientific programs that produce output data for use by interactive visualization engines.

The molecular dynamics simulation we are developing is one that permits human interactions not just with the output data produced by the code, but also permits interactions with the running program itself in order to effect on-line changes, called 'program steering' by the scientific community. For example, in our application, users wish to inspect the program's output data (i.e., images of the particles' locations or energy values) in order to modify program parameters like *timestep size* during program execution. This permits the program to move rapidly in simulation times of little interest and to increase fidelity at times of interest.

[1] Pipelining, in this case, refers to having a processor for each protocol layer and then allowing each layer to process a different packet. Therefore, for the OSI Model, up to seven packets are processed simultaneously, but each of these individual packets still sequentially pass through the same seven layers.

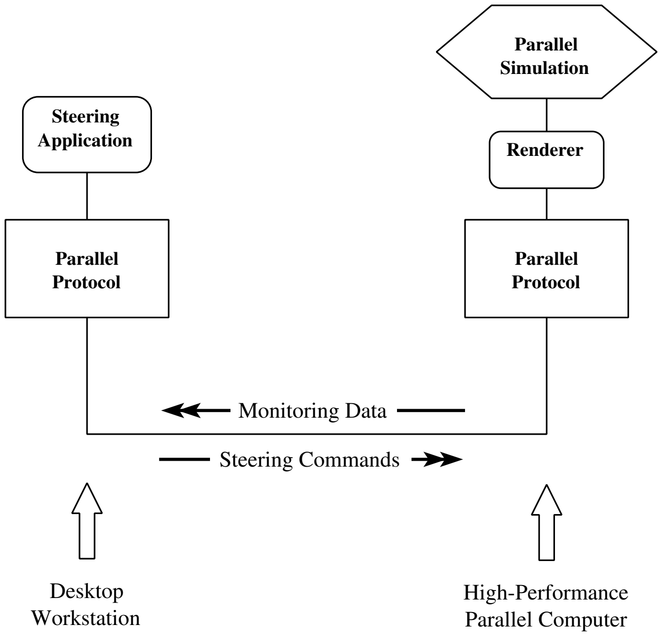

Figure 1: Relationship of a Simulation, an Online Steering Application, and the Parallel Protocol which connects them.

Furthermore, the human users of this program include both end users like physicists using the code for experimentation and developers interested in program performance.

The on-line use of both output data and performance information gives rise to the communication requirements depicted in Figure 1, which depicts bi-directional communications between the parallel supercomputer employed by the particle simulation program and the small-scale multiprocessor workstation (e.g., a multiprocessor Silicon Graphics machine) used for information display and for on-line steering. The interesting characteristics of such communication are: (1) traffic is unbalanced in that large amounts of simulation state flow from the supercomputer to the workstation, whereas small-scale traffic containing steering commands flows back to the simulation, (2) traffic from the supercomputer to the workstation consists of multiple connections that vary significantly in size and type ranging from (a) low-bandwidth connections for transmitting small amounts of performance monitoring information like 'processor utilization over time', to (b) moderate bandwidth connections for depicting animations of program functionality, to (c) high-bandwidth connections for the timely transmission of high-quality visualizations of program output data. Furthermore, all information used for program steering must be transmitted with low latency so that the user can control the running program based on recent data, whereas information used for backup files can be transmitted with higher latency.

Protocols used in interactive parallel programs must support the diverse communication requirements identified above. Furthermore, parallelism in protocol processing may vary in different machines, ranging from higher degrees of parallelism on the parallel supercomputer (in conjunction with the application parallelism on that machine) to smaller-scale parallelism employed on the visualization engine. These requirements give rise to three attributes of protocols for such heterogeneous supercomputer systems:

- Protocols must be configurable to multiple target machines, so that they can exploit the underlying network architecture and the diverse processor nodes, including the simultaneous use of multiple network devices on a single parallel machine able to transmit data to different user interface nodes (e.g., a visualization engine and a performance display node).
- Protocols should be able to support desired data processing in conjunction with network transmissions, such as data compression and/or encryption.
- Protocols must be able to support "standard" off-line file transfer traffic in conjunction with the real-time traffic described above.

3 A Prototype Parallel Communication Protocol

3.1 Protocol Functionality and Structure

This section presents a parallelized communication protocol possessing the features required for the distributed supercomputing applications described in the previous section. This protocol is implemented on a 32-node GP1000 BBN Butterfly shared memory multiprocessor. The purpose of this implementation is to evaluate the opportunities and challenges in parallel protocol processing on large-scale parallel machines (machines able to support more than 30 processors – the BBN Butterfly GP1000 can have up to 128 processors). In order to evaluate the overheads of parallel protocol processing, the prototype protocol's implementation actually performs both the send and receive processing on the same parallel machine, with the device simulated by one of the machine's processors. As a result, the effects of network routing, congestion, and latency are all avoided.

The prototype protocol consists of a set of protocol *objects*, each of which performs an isolated protocol processing task. As seen in Figure 2, each object contains a single or multiple threads of execution that perform processing specific to that protocol object. Objects communicate with each other by asynchronous invocations implemented as messages delivered via shared memory mailboxes, much like the implementation of object invocations in other

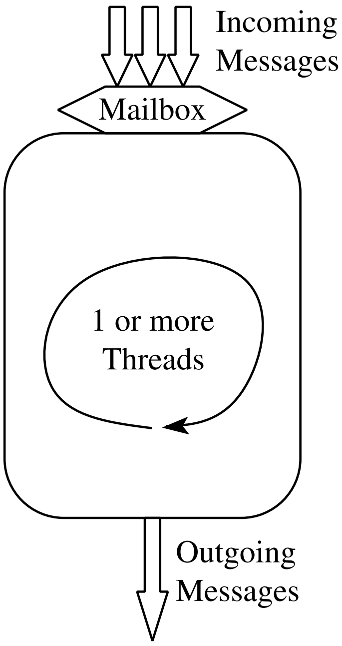

Figure 2: A Single Prototype Protocol Object

multiprocessor operating systems[Bihar-92, Chase-89]. Objects cooperate in the processing of protocol packets, and they may be composed to result in different communication protocol configurations in a manner more flexible than the layers of traditional protocol architectures allow. For instance, in the protocol in Figure 3, an outgoing user packet may be encrypted by the encryption object *while* the sequence numbers for the packet are computed by the reliability object. As a result, the fine grain objects in this protocol permit us to exploit any parallelism that might exist in a protocol's processing.

Figure 3 depicts the full structure of the prototype parallel protocol, including all of its objects and interconnections performing both the sending and the receiving tasks, and the simulation of the underlying network device. As apparent from the figure, protocol objects are functional units that perform separable tasks, such as interfacing with protocol users, controlling connections, processing incoming and outgoing packets, and encapsulating a physical interface. However, individual objects may contain internal parallelism, such as the encryption object implementing DES encryption. The collection of objects shown in Figure 3 provides reliable, connection-oriented transmission service between applications on the same local network with encryption and decryption. This results in service and functionality similar to typical *XNS*[2] protocols commonly in use today (except for routing). In addition, the prototype protocol uses both *go-back-N* or *selective reject* methods[3] to provide reliable,

[2]Xerox Network Service protocols and their derivatives including Novell IPX/SPX

[3]The method of error correction used may be specified by each application.

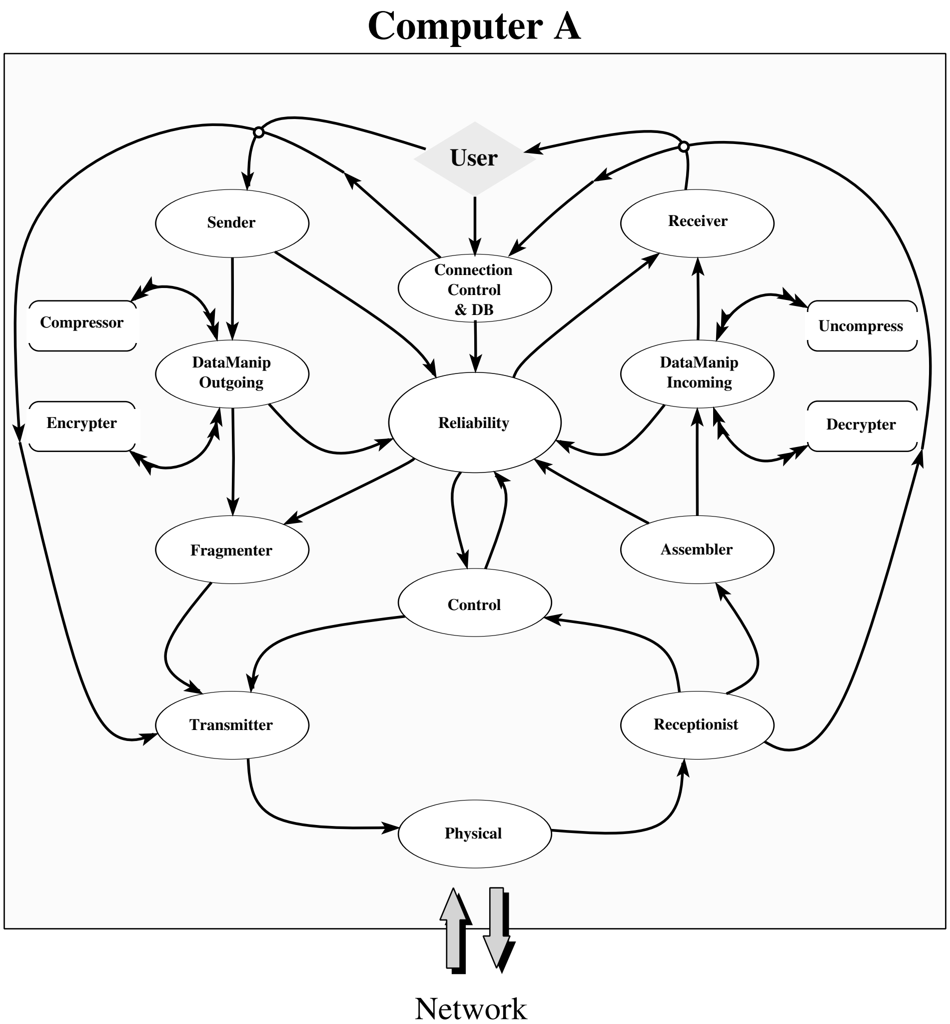

Figure 3: Protocol Objects and Their Invocation Structure.

connection-oriented service. The resulting protocol offers sufficient functionality for our purposes: the experimentation with parallelism in communication protocols.

3.2 Parallel Protocol Performance

All measurements reported below are attained on a 32-node GP1000 BBN Butterfly [Crow-85]. The Butterfly is a MIMD, shared memory parallel processor. Each processor node contains a 25Mhz Motorola MC68020 processor, a 68881 floating point processor, a 68851 Memory Management Unit (MMU), 4M bytes of RAM and a microcoded co-processor called the Processor Node Controller (PNC) which handles shared memory requests. Processor nodes are connected by a 32 megabits per second per path multistage switch which allows processor nodes to share their local memories with other nodes. For reference, a procedure call without parameters costs approximately 3 microseconds on the BBN Butterfly. The Mach operating system and a light-weight CThreads package developed by our group are being used[Schwa-91].

The protocol's implementation relies on a simple extension of CThreads with a mailbox communication facility. Each object represented in Figure 3 is implemented by at least one thread which accepts requests for invocation from other objects via its built-in mailbox, processes such requests, and then submits more invocation requests to other objects' mailboxes, as shown in Figure 2. Each request contains the parameters required by the target object and pointers to the message or message fragments being processed. Mailboxes themselves are represented as structures in shared memory.

In the remainder of this section, we first identify the opportunities for parallelism in communication protocols. Next, we describe the issues to be addressed in the efficient implementation of parallel protocols. These issues then give rise to the design of the protocol implementation framework described in Section 4.

Opportunities for Parallelism. The simple protocol described above offers the following opportunities for parallelism:

- Jain et al. demonstrated conceptually that *multiple packets in the transmission window can be processed independently* to significantly increase throughput[Jain-90]. This can be done without significant changes to our current protocol implementation.

- It is intuitive that *multiple connections can be processed in parallel*, since there are no direct dependencies between different connections. Therefore, the different threads associated with objects could be processing multiple connections simultaneously, or a single thread assigned to each connection could 'shepherd' each message through its required set of protocol objects (as done in the x-Kernel). Ideally, this should result in increases in protocol throughput linear in the number of processors and connections.

- *A single raw connection offers limited and scalable parallelism for each message being processed.*[4] As mentioned earlier, Clark, *et al.*[Clark-90] argue that data manipulation will be the rate limiting factor in data transmission. We hypothesize, therefore, that opportunities for parallelism on raw protocols are limited. Figure 4 depicts specific measurements demonstrating these limitations on the BBN Butterfly. The figure shows the total time required for sending 100 packets of 1k bytes each across a single raw connection, as the number of processors used by the objects in Figure 3 is increased from 2 to 10. In these measurements, the Physical object resides on its own processor, and all the other objects are spread uniformly across the available processors. The lower curve in the figure represents the execution time for processing all 100 packets without encryption, whereas the upper curve depicts execution time with serial encryption (i.e., encryption performed on a single processor).

  It is apparent that limited speedup is achieved for raw packets up to 6 processors. However, with more than 6 processors, the memory contention overhead starts to dominate, leading to substantial performance degradation for larger numbers of processors. Such contention arises from the fact that our initial implementation placed all mailboxes and their state variables into a single processor's memory, which leads to increased access rates and, therefore, access contention with larger numbers of processors. The improvements resulting from a more appropriate distribution of mailboxes (their location in memory local to their objects) are shown in Figure 5. This figure demonstrates that parallel communication protocols can be implemented such that they are scalable to large-scale parallel machines. In other words, while the parallelism in a raw protocol is limited, the use of additional processors in such protocols need not lead to excessive overheads and, therefore, performance degradation. This fact is important for the protocol implementations in conjunction with highly parallel application programs shown next.

  Finally, some of the overheads in our current implementation of the raw protocol stem from our mailbox-based implementation of invocations. Alternative implementation approaches for objects and object invocations such as the 'shepherding' done in the x-Kernel or the invocation primitives described in [Gheit-92] can further improve performance.

- *Opportunities for parallelism are improved substantially when protocol and presentation-level processing are both performed in parallel.* This is demonstrated by measurements depicted in Figure 6, which describe the total time taken to encrypt 100 packets of varying sizes, as the number of processors is increased from 1 to 10. These measurements show good speedups in protocol processing for larger packets and limited speedups for

[4] A raw connection is one that does not require any data manipulation like compression or encryption.

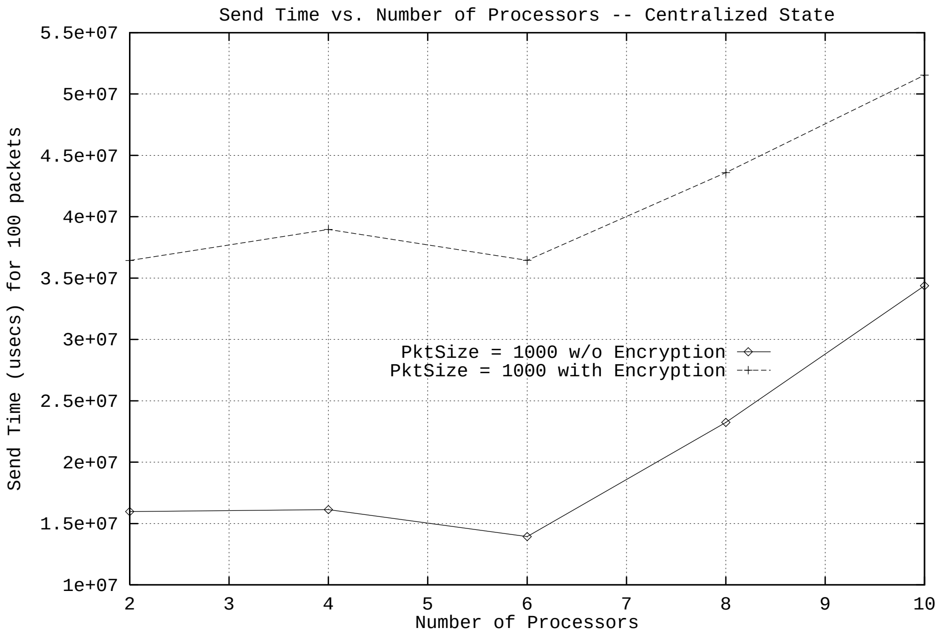

Figure 4: Protocol Processing Time with Varying Number of Processors (Centralized State)

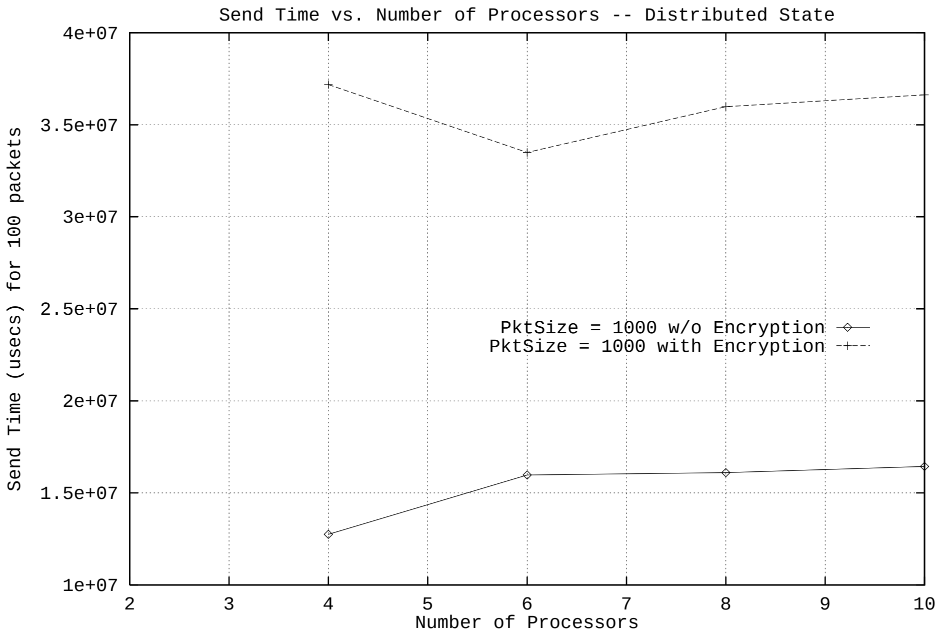

Figure 5: Protocol Processing Time with Varying Number of Processors (Distributed State)

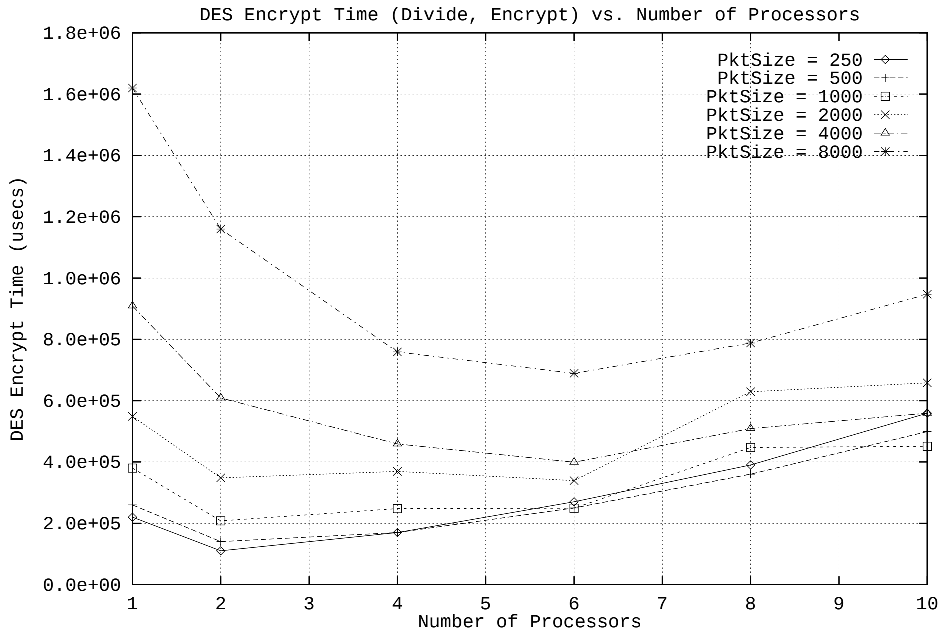

Figure 6: DES Encryption Time with Varying Packet Sizes

smaller packets. Differences in speedup for varying packet sizes are due to changes in the ratio of time spent operating on each sub-packet by each processor (which directly depends on sub-packet size) vs. the overheads due to the need to divide each packet into sub-packets being encrypted by each thread. Such overheads are discussed next.

Overheads in Parallel Communication Protocols: Data Movement and Remote Data Access. Parallel communication protocols are subject to the same performance penalties as other parallel application programs. These include concerns regarding the granularity of execution units, as demonstrated by the differences in speedup for the encryption of larger vs. smaller packets shown in Figure 6. They also include overheads caused by excessive data movement. Specifically, in parallel communication protocols, it is imperative that the actual data contained in each packet is moved as little as possible. Moreover, this property of a parallel protocol must be maintained even though different protocol objects may use different levels of parallelization and therefore, different message decompositions (e.g., encryption vs. compression). Additionally, the data submitted for send processing is likely to originate in a distributed form having been produced by different components of the parallel application (and therefore, from different memory units in the parallel machine).

We offer experimental evidence of the importance of minimization of data movement in parallel protocol processing. First, recall the centralized vs. distributed implementation of

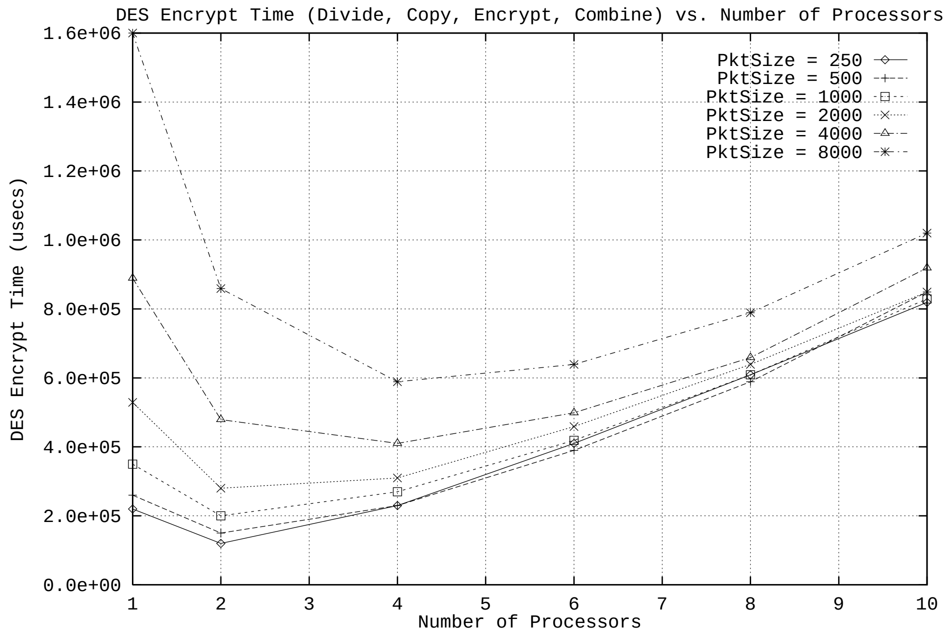

Figure 7: DES Encryption Time with 'Split-Join' Encryption Processing

mailboxes shown in Figures 4 and 5. These measurements demonstrate that excessive access to remote state caused by inappropriate state distribution will impede parallel protocol performance. Second, and more importantly, consider the parallel encryption object. When a message is delivered to this object in one piece, the object will first have to split the message into sub-packets for use by each thread performing encryption and then join the sub-packets into the larger message for use by other protocol objects. The overheads of such 'split-join' processing can substantially reduce protocol processing speeds. For our implementation, those overheads are depicted in Figures 7 and 8, for multiple packet sizes and for different numbers of processors. Specifically, in Figure 7, each packet is first split apart, copied locally, encrypted, then joined, whereas the last join step is not performed in Figure 8, presumably because subsequent objects can make use of the same submessage decomposition as that created by the encryption object. It is obvious from those figures that the additional memory copies required for message joining severely affect message processing speeds.

Note that some of the results in Figures 7 and 8 may change somewhat when using the block move facilities for data copying available in most modern parallel machines, including the BBN Butterfly. Our measurements of such tradeoffs on the BBN Butterfly basically demonstrate that penalties due to excessive data movement persist, when performing remote accesses to data or when copying the data using block move instructions.

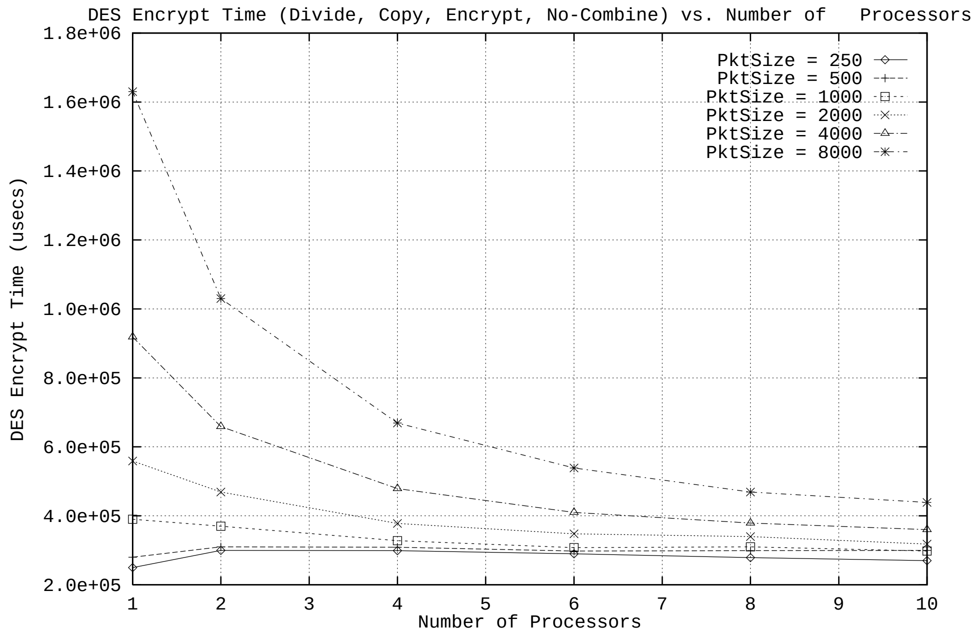

Figure 8: DES Encryption Time with 'Split' but Not 'Join' Processing

Performance Implications of Alternative Object Mappings. A second important topic regarding parallel communication protocols and other parallel programs is the mapping of protocol objects to processors. We demonstrate the effects of performance differences due to alternative mappings by use of arbitrary vs. rule-based assignments of protocol objects to processors. Our experimentation uses three mapping rules:

- Objects that must wait for each other should share the same processor;
- Sending and receiving sides of the protocol should be scheduled similarly; and
- Objects that touch every byte of packets should be located on the processor(s) where the data resides.

The effects of applying these rules to the prototype protocol (including encryption) are depicted in Figure 9, which shows the total send processing time for 100 packets, with increasing numbers of processors and alternative object-to-processor mappings. The two top curves in the figure depict measurements with 2k packets. The curve marked with square points shows performance when using the best rule-based mapping we could determine, whereas the curve with x-points assumes an arbitrary mapping of objects to processors (whatever is done by the underlying threads package). A similar trend is seen in the lower set of curves that depict performance differences due to mapping changes for 1k packets.

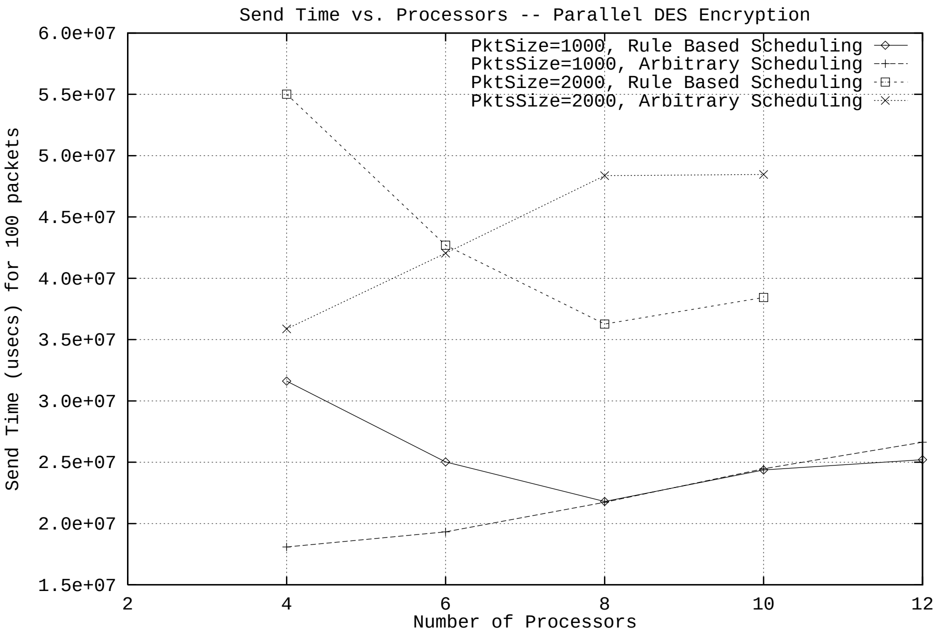

Figure 9: Processing Time with Different Object Mappings

The implication of the measurements shown in Figure 9 is that protocol programmers must be able to control and adjust the mapping of objects to processors for each path through the protocol, and typically, such a mapping is subject to change over time (as network traffic changes) due to differences in contention on specific protocol objects. This is our motivation for suggesting that any framework for protocol programming must permit programmers to implement protocol-specific policies of data (or message) caching and decomposition, as well as protocol-specific policies for object-scheduling.

3.3 Evaluation of the Prototype's Implementation

Several conclusions can be drawn from the initial implementation of protocol objects as defined in Section 3.1. We discuss these conclusions next, then use them to design the protocol programming framework described in Section 4.

Need for Multiple Invocation Semantics. A single semantics of object invocation is not sufficient for the efficient programming of parallel protocols. Specifically, with each asynchronous invocation implemented using mailboxes certain overhead is experienced. Such overheads can be avoided by offering an alternative synchronous invocation that may be used by a single thread to 'shepherd' a message through a short method of another object.

Additional invocation semantics, possibly involving priorities or deadlines, are necessary when extending our protocol to the real-time domain[Gheit-92].

"Done-with" Propagation. When a protocol object passes a copy of its packet or message fragment to another object, the space containing this copy cannot be reused until the target object completes its processing. Therefore, code must be written that notifies predecessor objects each time a single object is 'done with' a message or message fragment. Such code is not only prone to errors but it also gives rise to inefficiencies in protocol processing. As a result, a protocol programming framework should accommodate the flexible and efficient management of the state of multiple objects (much like exception handling in multiprocessor operating systems). Furthermore, the use of additional object invocations for this purpose is not appropriate since the accompanying overhead appears excessive compared to the overhead of a facility dedicated to such state management.

Dynamic Object Interconnection. It should be possible to change object connections (the invocation structure) dynamically. Otherwise, inefficient protocol implementations result for several reasons. First, if the addresses of target objects are hardcoded into object methods, then an object like Sender must always send its packet to DataManipulatorOutgoing (see Figure 3), even if the particular packet's connection required no manipulation. In addition, programmers are forced to implement proxy objects like DataManipulatorOutgoing, which exist only to determine whether the actions of their associated objects, like Compressor and Encrypter are actually needed for the message in question. Similarly, if programmers wish to design multiple implementations of a single object, such as a reliability object offering 'go-back-N' and another reliability object offering 'selective-reject,' static object interconnection tends to prompt programmers to implement a single, complex object offering both options (in order to avoid proxies).

Dynamic Object Instantiation. It would be useful to be able to dynamically instantiate additional objects to service unexpected traffic. For example, under conditions of heavy load, we might create an additional Encrypter object that exclusively services high priority packets, thereby able to retain some guarantees of message processing time for certain sets of packets.

Internal Object Configuration. The initial protocol design assumed that each protocol object is a black box exporting only its methods and method parameters to other objects. This assumption is not suitable for high-performance parallel protocols for two reasons: (1) certain parameters or 'attributes' of each object must be related to those of other objects (e.g., object location, desired fragment size, etc.) and (2) objects share significant amounts of state (e.g., the 'done-with' state). It is therefore imperative that objects offer a standard means for specification and on-line change of such object attributes not captured by their standard interfaces. One example of a desirable dynamic change to such an object attribute would be a change in the compression object's compression factor, which could be used to realize tradeoffs in processing time vs. required network bandwidth.

4 A Framework for Programming Parallel Communication Protocols

The previous section identified several critical characteristics of any framework for the implementation of efficient parallel communication protocols. They included: (1) internal object configuration permitting both single and multi-threaded objects and permitting the dynamic control of object location, message decomposition, and the resulting movement of message fragments, (2) dynamic object and connection instantiation, and (3) the need for multiple invocation semantics (e.g., synchronous and asynchronous invocations), including the maintenance of state shared among multiple objects (e.g., 'done with' propagation). In general, the protocol framework must permit the construction of protocols that are easily tailored to different numbers of processors using varying object-processor assignments, thereby facilitating the adaptation of protocol processing to the available parallelism in the application program using the protocol.

We accommodate the needs summarized in the previous paragraph by defining an *object* in the Protocol Programming Framework (PPF) (see Figure 10) as an encapsulation of some state or behavior. As with objects in all object-based environments, a PPF object exports object-specific methods that access its state or invoke its functionality. However, in contrast to these environments, the PPF defines additional object methods used for object configuration, connection setup, and memory management. Lastly, the PPF provides three "utilities" that aid in the implementation of an object's exported methods: Alarms, Hash Tables, and Memory Objects.

Global Object Names. In the PPF, a method exported by a object has a name, parameters, and three additional attributes: the number of required *invocations*, a *done-with* behavior, and an invocation *type*. Method names are global names so that objects can be interconnected statically and dynamically. For example, in Figure 11, the Fragmenter's "outbound fragments" outlet has been connected to Physical's "transmit" method.

Event-based Method Activation. Objects often require that multiple inputs be provided before method execution can commence. For instance, a fragmenting object must have both the packet's header *and* its data before proceeding with fragmentation. When protocol objects are executing in parallel, the simultaneous arrival of both at the same target object is not trivially guaranteed. As a result, the PPF uses triggers inside objects that prevent the execution of an object's internal functions unless all input requirements are satisfied. For example, one useful trigger in the Fragmenter would be the requirement that the object be invoked twice for each packet before the internal `fragment()` function is executed.

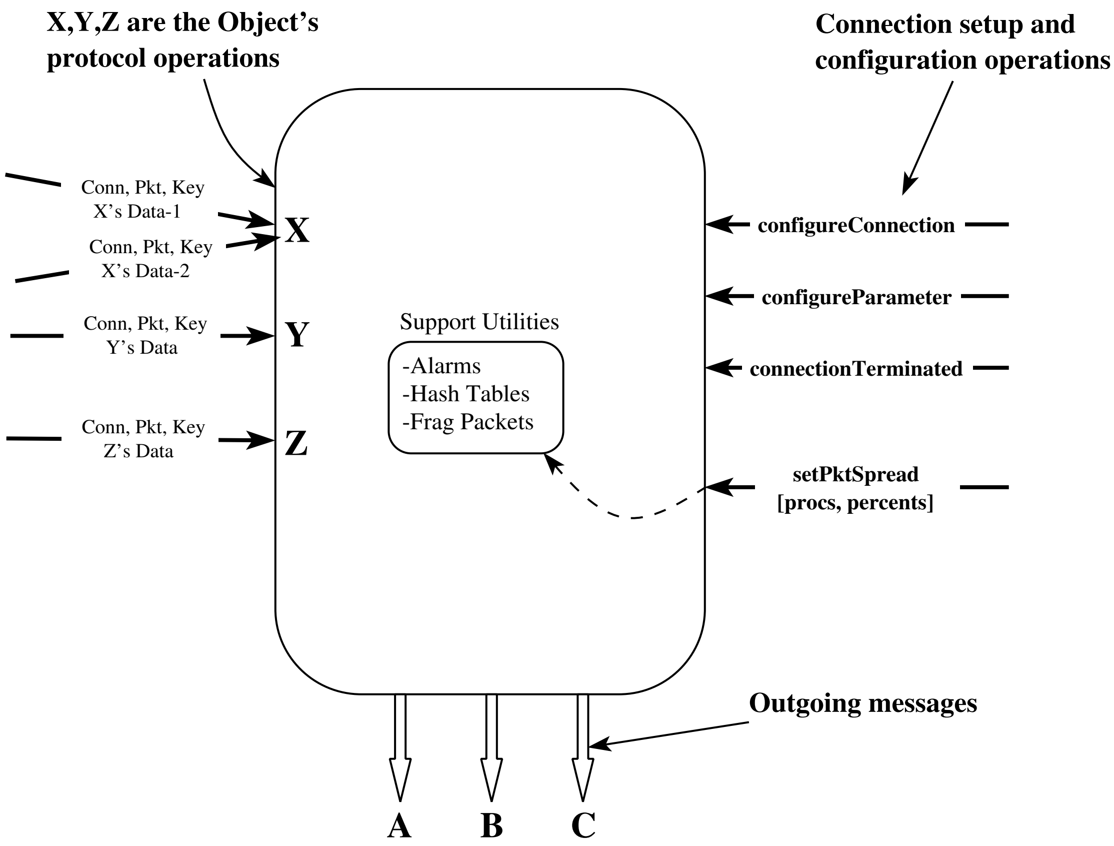

Figure 10: Structure of a Protocol Object.

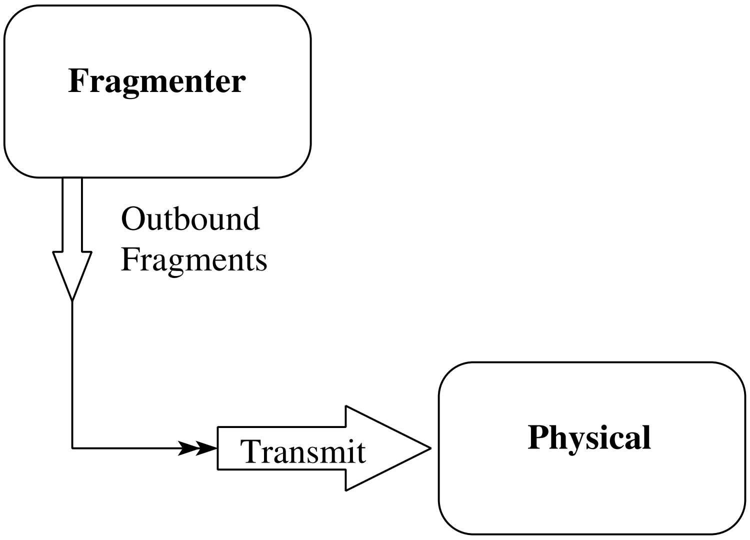

Figure 11: Objects are Connected by Linking an Object's Outlet to Another's Exported Method.

Default Methods. The PPF will add default methods to each object instance, including the 'done-with' methods mentioned earlier. This is done in two steps. First, as state is maintained and pointers to messages are passed between objects, the PPF maintains information regarding the 'trail' of objects through which each message has passed. Whenever an object that neither stores local state about a message nor passes message pointers to other objects completes its execution, the PPF notifies the preceding objects in the message's 'trail' (in reverse order of their original invocations), thereby causing them to clean up their internal states. As a result, programmers do not have to code explicit state maintenance. As an example, again consider the Fragmenter and Physical objects. In this case, the Fragmenter first accepts a large packet, then breaks it into several smaller pieces, and finally forwards pointers to each to Physical. Physical does not store any state after it finishes transmitting, so it automatically performs a "Done-with." Therefore, as soon as the Transmit method completes, Fragmenter's Done-with method is automatically invoked.

Multiple Invocation Semantics. With each method invocation, an additional 'type' parameter specifies the method implementation as 'synchronous' or 'asynchronous', resulting in the execution of different invocation code. The efficient implementation of such alternative invocation primitives is described in detail in [Gheit-92]. Additional invocation semantics may be added as needed.

Object and Connection Configuration. In order to custom-fit protocols to the application programmer's requirements and to the network's current status, objects and their connections may be configured at compile- or at runtime. Runtime configuration makes use

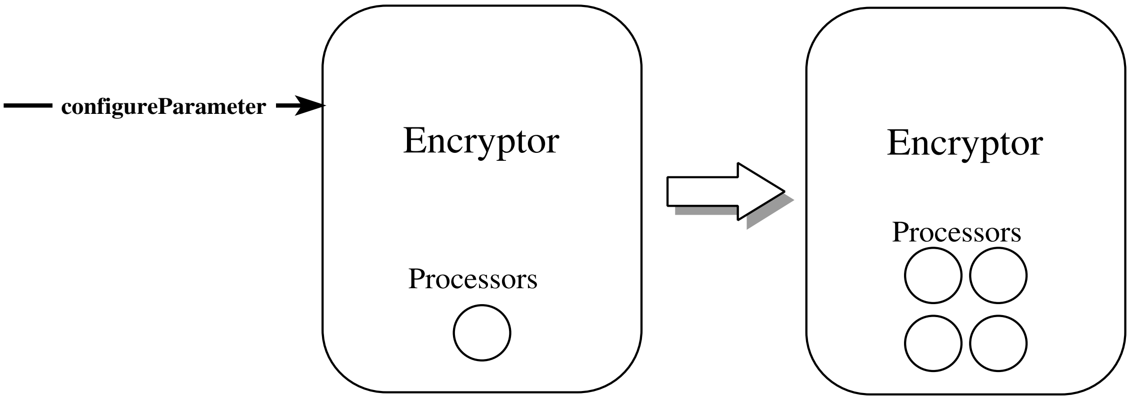

Figure 12: The ConfigureParameter Method can Vary the Parallelism Used for Each Connection.

of additional methods associated with each object. Specifically, each protocol object may optionally export attributes that may be set by other objects. These attributes are modified using the `configureParameter` method exported from each PPF Object. For instance, an Encryption object may be configured such that its assignment of threads and methods to processors uses a different processor for each established connection vs. using multiple processors in the encryption of a single, decomposed message (see Figure 12). We call such configuration "internal" object configuration.

"External" protocol configuration involves manipulating the interconnections of protocol objects. For example, when an application requests that compression be enabled, a Compression object will be inserted into the protocol graph with a `configureConnection` operation (see Figure 13).

An issue with runtime configuration not addressed in this paper is the determination as to 'when' such configuration should occur, such as 'at the time of connection establishment,' 'during the use of a single connection,' etc. Our future work will initially consider connection-time configuration, but we will implement the PPF such that on-line status collection is available for more dynamic configuration changes.

Support Utilities. As with the x-Kernel's micro-protocols, the PPF will offer facilities for implementing the following standard functionality: Alarms, Message Manipulation, and Hash Tables, each of which are discussed next. In contrast to the x-Kernel, the PPF's design and implementation of both is tailored both to the PPF object model and to efficiency on multiprocessor architectures.

When an object starts an Alarm timer, it provides a method that it wishes to be called in the future and is returned an alarm handle. Using this handle, the object can cancel the Alarm when it's no longer needed. In this way, the PPF can not only efficiently service

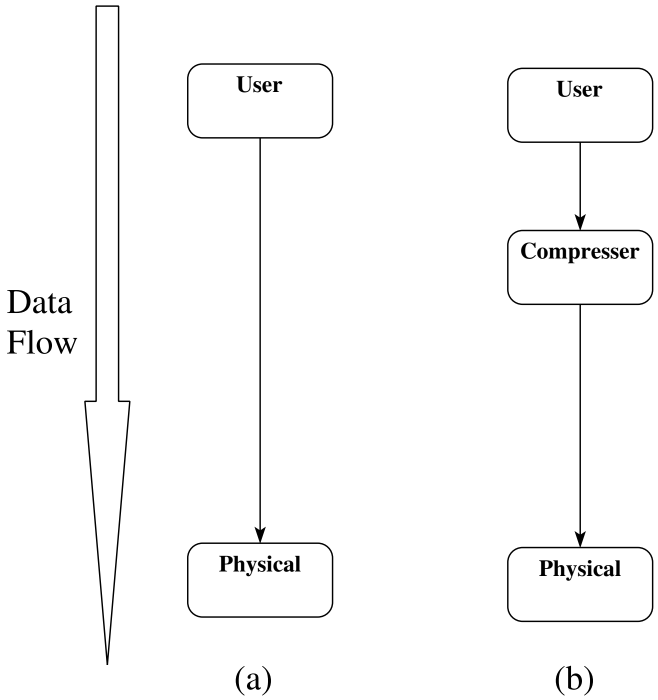

Figure 13: The ConfigureConnection Method Can Be Used to Insert a Compressor Object into the Outgoing Packet Stream.

expiring Alarms but can also quickly cancel them, as alarms are more often cancelled than expired in modern networks.

Secondly, protocols typically need to store significant amounts of information which need to be found quickly as packets are serviced. For instance,

- Each connection's state must be found to determine both protocol parameters, e.g., the encryption key or the number of compression processors, and the objects that are to receive the packet next (e.g., fragmentation follows compression for connection 1 but follows encryption for connection 2).
- A windowful of packets may be in process for each connection, and PPF objects that maintain state must know where to store information about their "current" packet.

To make these mappings as efficient as possible, they are done only once: each object creates an array(1..MAX_CONNECTIONS) which it uses to store connection state. When a new connection is requested, a handle is assigned to the application, and an available number is internally assigned and passed along with each packet. Therefore, each object need only look in its array (handle) to find its internal state. Similarly, as multiple packets are sent in a single window, the PPF, upon entry of packets into the protocol, labels each one by its window position. In this way, only the framework (and not each object) worries about connection-id and window-position mappings.

Lastly, the experiences with parallel communication protocols reported in Section 3 demonstrate that parallel message processing requires a message structure that:

- Supports packets whose data is divided among several processors;
- Minimizes data movement, particularly across processors;
- Avoids rejoining of packets' contents after they have been divided;
- Can adapt *itself* to objects' locations; and
- Takes advantage of prefetch and block-move mechanisms available on most shared memory multiprocessors.

PPF's fragmented packet mechanisms attempt to satisfy these needs as follows. Specifically, each protocol object which manipulates packets' contents specifies either one or both of the following:

- The preferred atomic size of packets, e.g., an ethernet fragmenting object would specify that 1500-byte-multiple packet sizes are best (1500, 3000, ...), or

- The processors that it will use to process the packets and, therefore, those on which it would like the packets.

Given these hints, the PPF will use the protocol objects' `setPacketSpread` method to propagate the information to parent objects' fragmented packet libraries. Last, when these parent objects request packet blocks, the library will use its knowledge of the needs below to spread the packets across the appropriate processors.

5 Concluding Remarks

This paper demonstrates opportunities for parallelism in communication protocols. Experimental results attained on a 32-node BBN Butterfly multiprocessor provide evidence of useful parallelism in the processing required for a single connection, especially when including data manipulation by an application, such as encryption or compression. Experimental results demonstrate several factors critical to achieving high performance, including (1) the need to minimize data movement and remote data access both to state internal to the protocol's implementation and to the data contained in messages and passed to the communication protocol's components, and (2) the importance of appropriate mappings of protocol components to processors performing the various tasks part of each protocol's functionality. The main insights concerning protocol implementation resulting from these measurements are:

- Substantial parallelism exists in communication protocols, but it must be exploited in conjunction with the parallelism inherent in the application programs using the protocol. Otherwise, needless overheads may arise from data movement required by interactions of protocol components with application code. More specifically, we must perform both (1) the mapping of application and protocol code to different processors and (2) the decomposition of message data by the application and in the parallel protocol such that data movement is minimized and such that parallelism among protocol components is maximized.

- The cost of collecting data originating on multiple processors into a single message sent to a single network device can be prohibitive on large-scale parallel machines. This implies that single large-scale machines (like the Kendall Square Research supercomputers) should use multiple communication devices attached to different processors of the machine (much like manufacturers are now routinely attaching multiple disks to such machines).

Additional insights from our work concern the utilities needed for programming high-performance communication protocols. Our group is now developing a protocol programming framework that uses object-based representations of protocol components. This framework will be the basis for the development of parallel communication protocols for machines ranging from small-scale multiprocessors to large-scale shared memory machines. Some of the interesting research issues we are addressing include: support for connection-specific caching of message fragments, dynamic protocol configuration in response to application needs or network status, and the enforcement of real-time constraints in protocol processing.

Some limitations of our current results are derived from the simulation of actual network devices (since we are primarily interested in the protocol processing overheads) and from the relative simplicity of the current protocol implementation. These limitations will be addressed by the next steps of this research, which are targeting the implementation of a parallel communication protocol on a pair of parallel machines, across a high-performance network link (an FDDI or a HIPPI link) for the supercomputer application described in Section 2 of this paper.

Acknowledgements

The authors wish to thank Philip Chang and Stephen Chou for their efforts during our conversations and the prototype's development.

References

[Abbot-92] Mark B. Abbot and Larry L. Peterson, "A Language-Based Approach to Protocol Implementation," *ACM SIGCOMM-92 Symposium*, Baltimore, MD, Oct. 1992, pp. 27-38.

[Atwoo-91] J.W. Atwood and W. Prager, "A STREAMS-based Communications Subsystem in Turing Plus for Protocol Development," *Software–Practice and Experience*, Vol. 21 (December 1991), pp. 1365-1380.

[Bihar-92] Thomas E. Bihari and Karsten Schwan, "Object-Oriented Real-Time Systems: Concepts and Examples," *IEEE Computer*, Vol. 25 (Dec. 1992), pp. 25-32.

[Bihar-91] Thomas E. Bihari and Karsten Schwan, "Dynamic Adaptation of Real-Time Software," *ACM Transactions on Computer Systems*, Vol. 9 (May 1991), pp. 143-174.

[Chase-89] Jeffrey S. Chase, Franz G. Amador, et al., "The Amber System: Parallel Programming on a Network of Multiprocessors," *Twelfth ACM Symposium on Operating System Principles, SIGOPS Notices 23, 5*, 1989, pp. 147-158.

[Clark-89] David Clark, Van Jacobson, *et al.*, "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine* (June 1989), pp. 23-29.

[Clark-90] David Clark and David Tennenhouse, "Architectural Considerations for a New Generation of Protocols," *ACM Computer Communications Review*, Vol. 20 (September 1990), pp. 200-208.

[Crow-85] W. Crowther, J. Goodhue, et al., "The Butterfly Parallel Processor," *IEEE Computer Architecture Technical Committee Newsletter*, Dec. 1985, pp. 18-45.

[Dean-90] S.M. Dean, D.A. Kumpf, and H.M. Wenzel, "CONE: A Software Environment for Network Protocols," *Hewlett-Packard Journal* (February 1990), pp. 18-28.

[Dinge-90] J. Dinger, K. Rothermel, and K. Urbschat, "ECSE: An Efficient Environment for Layered Communication Protocols," *Second IEEE Workshop on Future Trends of Distributed Computer Systems*, Cairo, Egypt, 1990, pp. 305-314.

[Doeri-90] Willibald A. Doeringer, Doug Dykeman, em et al., "A Survey of Light-Weight Transport Protocols for High-Speed Networks," *IEEE Transactions on Communications*, Vol. 38 (November 1990), pp. 2025-2039.

[Garg-90] Arun Garg, "Parallel STREAMS: a Multi-Processor Implementation," *USENIX* (Winter 1990), pp. 163-176.

[Gheit-92] Ahmed Gheith and Karsten Schwan, "CHAOS: Kernel Support for Multi-Weight Objects, Invocations, and Atomicity in Real-Time Multiprocessor Applications" (unpublished Technical Report: Georgia Institute of Technology, 1992).

[Haas-91] Zygmunt Haas, "A Protocol Structure for High-Speed Communication over Broadband ISDN," *IEEE Network Magazine* (January 1991), pp. 64-70.

[Hutch-89a] N.C. Hutchinson, S. Mishra, *et al.*, "Tools for Implementing Network Protocols," *Software–Practice and Experience*, Vol. 19 (September 1989), pp. 895-916.

[Hutch-89b] N.C. Hutchinson, L.L. Peterson, and Herman Rao, "The *x*-Kernel: An Open Operating System Design," *IEEE Proceedings of the Second Workshop on Workstation Operating Systems*, Pacific Grove, CA, September 27-29, 1989, pp. 55-59.

[Hutch-89c] N.C. Hutchinson, L.L. Peterson, *et al.*, "RPC in the *x*-Kernel: Evaluating New Design Techniques," *Operating Systems Review*, Vol. 23,5 (Proceedings of the 12th ACM Syposium on OS Principles, Dec. 3-6, 1989), pp. 91-101.

[Jain-90] Niraj Jain, Mischa Schwartz and Theodore Bashkow, "Transport Protocol Processing at GBPS Rates," *ACM Computer Communications Review*, Vol. 20 (September 1990), pp. 188-199.

[Kanak-88] Hemant Kanakia and David R. Cheriton, "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors," *ACM SIGCOMM '88 Symposium*, Stanford, CA, Aug. 16-19, 1988, pp. 175-187.

[Krish-89] A.S. Krishnakumar and Krishnan Sabnani, "VLSI Implementations of Communication Protocols–A Survey," *IEEE Journal on Selected Areas in Communications*, Vol. 7 (September 1989), pp. 1082-1090.

[Netra-90] A.N. Netravali, W.D. Roome, and K. Sabnani, "Design and Implementation of a High-Speed Transport Protocol," *IEEE Transactions on Communications*, Vol. 38 (November 1990), pp. 2010-2024.

[Peter-90] L.L. Peterson, H.C. Hutchinson, *et al.*, "The *x*-kernel: A Platform for Accessing Internet Resources," *IEEE Computer Magazine* (May 1990), pp. 23-33.

[Preso-92] D.L. Presotto, "Multiprocessor Streams for Plan 9" (unpublished).

[Sabna-89] K. Sabnani and A. Netravali, "A High Speed Transport Protocol for Datagram/Virtual Circuit Networks," *ACM SIGCOMM '89 Symposium*, Austin, TX, September 1989, pp. 146-157.

[Schwa-91] Karsten Schwan, Harold Forbes, et al., "A C Thread Library for Multiprocessors," Georgia Institute of Technology Tech Report GIT-ICS-91/02, Jan. 1991.

[Watso-87] R.W. Watson and S.A. Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices," *ACM Transactions on Computer Systems*, Vol. 5 (May 1987), pp. 97-120.

[Zitte-91] M. Zitterbart, "High-Speed Transport Components," *IEEE Network Magazine* (January 1991), pp. 54-63.