The Design and Evaluation of an Off-Host Communications Protocol Architecture

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science (Computer Science)

by

Jeffrey R. Michel

August 1993

APPROVAL SHEET

This thesis is submitted in partial fulfillment of the
requirements for the degree of
Master of Science (Computer Science)

_____
Author

This thesis has been read and approved by the Examining Committee:

_____
Thesis Advisor

_____
Committee Chairman

_____

_____

Accepted for the School of Engineering and Applied Science:

_____
Dean, School of Engineering and
Applied Science

August 1993

Abstract

The University of Virginia Computer Networks Laboratory has developed an implementation of the SAFENET lightweight suite of communication protocols using an off-host implementation of the Xpress Transfer Protocol (XTP). We used an attached protocol processor to test the hypothesis that such an architecture could simultaneously optimize protocol performance (by giving it a dedicated coprocessor) and user application performance (by freeing the host from the burden of protocol processing). Our experience indicates that the choice of implementation architecture has a profound impact upon the overall system performance. This work surveys the design issues inherent to off-host communications architectures and discusses the design choices made in our own architecture. Also presented are our performance results and the many lessons learned from the analysis of our implementation architecture. We also provide a survey of related work performed by other researchers. To assist the designers of future off-host architectures and to evaluate the degree to which we exploited our particular architecture, we develop a simple analytic model to predict the performance of an off-host architecture using readily-obtainable input parameters. The predictions of the model as applied to our system are compared with the system's observed performance.

Acknowledgments

Many deserve appreciation for helping to bring this work to fruition. First of all I would like to thank our sponsor, SPAWAR, for funding this research. Further, I wish to acknowledge my colleagues Robert Simoncic, Bert Dempsey, John Fenton, and Alex Waterman for their invaluable assistance on our project. In particular, I wish to thank Alex for generously providing preliminary results from his on-host XTP implementation. I am also indebted to Craig Meyers for his extensive cooperation and Bob Ross for his hardware expertise. Thanks are especially due to my advisor, Alfred Weaver, whose wisdom and guidance have allowed me to make achievements here at the University of which I can be proud. Above all, I wish to thank my parents for their endless support, which has made all of this possible.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1. Background

Although modern local area networks (e.g., FDDI, desktop ATM, and HiPPI) promise both high performance (100 Mbps or more of throughput) and new functionality (synchronous bandwidth), traditional computer system and application designs—those that have functioned well for older networks (e.g., Ethernet) operating asynchronously and at lower data rates—have had difficulty delivering on such promises. Since the modern technology brings significantly-increased bandwidth into which computer systems may tap, one might naively assume that similar improvements occur in throughput from the perspective of application processes. Recent measurements, however, show that this is not the case. For example, although FDDI provides a transmission rate ten times higher than that of Ethernet, the throughput available to an application process using FDDI is typically only double or triple that available from Ethernet.

The reason for such problems becomes apparent when one recognizes that between the high-speed network medium and an application process stands an architectural gauntlet of computer system hardware, communication protocols, and operating system software. As system designers have attempted to supply more of the growing network-media bandwidth to application programs, they find that these architectural fixtures are becoming more and more of a bottleneck. In particular, it has been observed that the execution of communications protocols has played a major role in constraining user application throughput. The research in this thesis is concerned with attacking the problem of protocol execution.

The recent criticism of protocols has prompted a rethinking of their implementation architectures. Although many variations exist, there are only two *basic* choices for where to execute protocols; either application processes and protocols must share the host processor and contend for its cycles, or the protocols must run off-host on dedicated coprocessor hardware and pay the penalty for the communication between their processor and the host

computer system. The former represents the classical approach (e.g., TCP/IP embedded in a UNIX kernel) and has been well studied; the latter is novel and has not been the subject of as much academic investigation. At least in theory, the off-host approach seems to be a route for improved performance. By off-loading the host of its communications processing responsibilities, application processes will have more host processing cycles available, and by giving communications protocols their own processor, their implementations may be tuned to provide better performance to applications. In addition, the dedicated cycles which the protocol processor provides to protocol processing may be essential for maintaining the real-time communications processing deadlines inherent in continuous-media applications that make use of synchronous bandwidth. It is reasonable to expect, however, some additional sources of overhead with the off-host approach due to the complexity of its architecture. The identification of this overhead and its minimization through proper design choices are principal subjects of this thesis.

The goals of this thesis are: (a) to identify the general design issues inherent in off-host communications protocol architectures, (b) to study an actual off-host protocol architecture, presenting its design, implementation, and performance, (c) to identify the performance bottlenecks in the architecture, relating them to architectural considerations in its design, (d) to develop an analytic model of the performance of off-host communications architectures and to validate it through comparison with observed performance, and (e) to make qualitative judgements about the suitability of off-host communications architectures.

1.2. A General Model of Communication

Before delving into the issues of off-host protocol implementations, we first present an overview of our general model of computer communication, including the topics of computer networking and communication protocols. This prepares us for the discussion of communication architectures employing off-host protocol implementations. A *computer*

network consists of a set of geographically-distributed, autonomous computers, termed *hosts*, that are connected via communications links. Physical links between the individual hosts provide the basic ability to communicate through the modulation of fundamental physical signals such as electrical potential differences and electromagnetic radiation. In addition to this physical media, certain conventions are needed to provide a discipline for the use of the media and to give meaning to the signals it carries. These conventions are expressed in the form of *protocols*, sets of rules governing issues such as the physical links themselves, the interaction between network hosts, and the syntax and semantics of the information they transmit and receive.

These communications protocols are quite numerous and have perform varying services. To organize this complexity, they are structured into *protocol hierarchies* consisting of multiple layers of functionality. For the purposes of our discussion, we shall use the hierarchical organization of the ISO/OSI reference model [24] to refer to the protocol layers and to imply the functionality supported at each. This seven layer model, illustrated in Figure 1.1, is structured such that a protocol at layer *n* in the hierarchy employs the services of the protocols at layer (*n* - 1) to provide communications services to the protocols at layer (*n* + 1). Below the first layer are the physical communications media, and above the seventh layer are application processes.

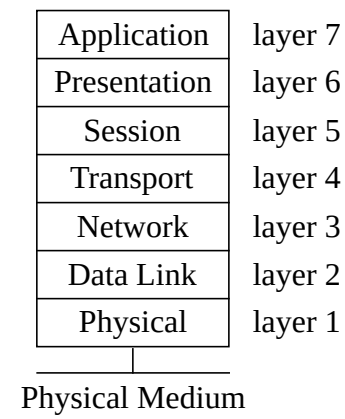


Figure 1.1: The ISO/OSI Reference Model

A brief summary of the functionality of the protocols present at each layer in the OSI reference model is as follows.

- An *application layer* protocol is responsible for dictating the format of distributed data structures and defining the sequence of data unit exchanges for a given type of application. The protocol directly serves the application programs above it.

- A *presentation layer* protocol provides a standardized syntax for application layer data which preserves the semantics of the information, even in the face of host system heterogeneity. Operations such as encryption may also be done at this layer.

- A *session layer* protocol exists to organize and synchronize presentation layer data units. Connection establishment and release may be done at this layer.

- A *transport layer* protocol is responsible for the end-to-end transfer of data units with certain reliability semantics from a session layer protocol in one host to a peer protocol in another. Issues such as multiplexing data units from several user processes onto a single network are often handled here. In addition, segmentation and reassembly of data units and error and flow control are often performed at this layer.

- A *network layer* protocol provides the means to route a transport layer data unit across subnetworks and through intermediate hosts to its final destination host. One of several access points to separate network links may be chosen at this layer in order to forward a data unit to its proper destination.

- A *data link layer* protocol enables the exchange of framed network layer data units between adjacent hosts on a multi-access or point-to-point network link. Various reliability semantics and error detection are often supported here.

- A *physical layer* protocol provides the ability to transfer the bits of data link protocol data units on the underlying physical medium. It specifies the mechanical and electrical standards of the medium as well as the signalling techniques involved.

Taken together, these protocols exist to provide communications services to application programs. The *data path* of a data unit sent from one application process to another starts at a sending process that submits data to the application layer. The data descends through all seven layers of the model and onto the physical medium. From there it propagates through the network, possibly ascending and descending through the protocols at layers one through three of the reference model in intermediate hosts. The data eventually reaches the receiving host, where it ascends all seven protocol layers before reaching the receiving application process.

We refer to a collection of communications protocols organized into a stack covering each layer of the reference model as a *network protocol architecture*. In practice, one protocol may span multiple layers and some layers may be null. An example of a protocol architecture is the SAFENET architecture, which is discussed further in chapter four. A subset of the protocols in a protocol architecture that still covers each layer of the reference model is termed a *protocol suite*. An example of a protocol suite is the lightweight suite of SAFENET, which is also discussed in chapter four.

1.3. Protocol Implementation Possibilities

In this work we shall be concerned not with the details of the protocols residing at the layers of the reference model, but rather with their *implementation*. Particularly, we shall study the implementation of the transport and network layer protocols. When a protocol suite is to be realized in terms of an actual computer system and physical network hardware, we must concern ourselves with the production of each of its protocol layers. We introduce the term *communications architecture* to refer to the realization of an entire suite of protocols, serving a host computer system. Some components of the communications architecture may be built in host software, while others may be constructed off-host in additional hardware or firmware.

In the early history of computer networking, all but the physical layer communications protocols were implemented in the host computers, i.e., they were built in software executed by the host's central processing unit (CPU). Since physical layer protocols must modulate the physical medium and dictate mechanical standards, such protocols were necessarily implemented using special hardware devices. As time progressed, much of the functionality of the OSI data link layer has been implemented in additional VLSI hardware circuitry complementary to the host CPU [4][19][42]. The goal of this hardware has been twofold: (1) to perform protocol operations at generally higher rates of speed, and (2) to lower the burden of protocol processing on the host CPU [28].

The trend has been for an increasing amount of higher-layer protocol functionality to be performed in special-purpose hardware without the involvement of the host CPU. Following this trend, we investigate the implementation options for protocols at the network and transport layers. For the remainder of this work we assume that the physical and data link protocols are implemented on an integrated hardware device, hereby referred to as a *network interface*. In addition, protocols present at or above the session layer of the reference model are assumed to be implemented in host software and are not considered in any further detail. In the following sections we contrast the in-host versus off-host approaches to protocol implementation.

1.3.1. On-Host Implementation

The traditional approach to transport and network layer protocol implementations has been to place them in software on the host platform and execute all protocol operations using the host CPU. Once the choice has been made to use an on-host implementation, there are actually several possibilities concerning where its software may reside within the host. The choice of where to place the protocol implementation will likely be dictated by operating system architecture and security concerns.

In single-user or embedded computer systems, the secure use of shared resources between multiple processes is not typically a concern. Furthermore, there is seldom any address space partitioning between user processes and the operating system. In this environment, a protocol implementation may acceptably be in the form of library code linked with the code of an application program [41]. This approach has the benefit of avoiding operating system security concerns and complex address space issues. As a result, the protocol implementation is likely simpler and its performance is typically higher than that of other approaches. For example, with this implementation architecture, protocol services can be invoked via the relatively efficient mechanism of subprogram calls, thus promoting a low end-to-end latency.

In systems with multiple processes or multiple users, shared resources often need to be managed in a more secure fashion. This is typically accomplished through the use of an operating system kernel residing in a protected address space. In such situations, transport and network protocol implementations reside in the kernel. The provision of security is necessary because such protocols may use special I/O address spaces, shared network resources, or shared data structures that must be kept with some degree of integrity.

In this kernelized scenario, application programs typically request protocol services from the kernel through system calls. The protocol software may be loosely integrated into the kernel as a device driver accessed through the operating system's I/O system calls [37], or it may be tightly integrated into the kernel and accessed with its own set of networking system calls [30]. Such calls are more sophisticated than subprogram calls; hence, they may result in a significant overhead. In addition, if the kernel resides in a protected address space, the cost of some means of accessing protocol data residing in the calling process' address space must be incurred. An additional drawback to the kernelized approach is that it complicates the software of the kernel.

A hybrid approach is to use a privileged server process, referred to here as a *protocol server*, to contain the protocol implementation. In such a situation, an application process desiring network service acts as a client, sending a request message for protocol service to the server via interprocess communication. The server process later sends a reply message to the client when it has completed the request [26]. Access to the shared resources is restricted to the server process, which, like the kernel, has a protected address space. This approach has the benefit of providing secure access to shared resources without adding to the complexity of the operating system kernel. In addition, no special system calls are required for network communication. It is not clear, however, whether the overhead of system calls is reduced or increased by this scheme. Furthermore, a unique source of overhead due to the protocol server approach is that process context swaps must occur between the client and server processes.

No matter how the code of the protocol implementation is distributed in the host software, the host CPU has the sole responsibility of performing all protocol operations. The options mentioned vary only in the means in which higher layer protocols communicate with the protocol implementation. The actual approach to developing protocol software, once its in-host placement is determined, has been well studied.

1.3.2. Off-Host Implementation

Recent efforts have resulted in the implementation of network and transport layer protocols *off-host*. That is, such protocols are executed off-host by additional hardware rather than by the host's CPU. We refer to this additional hardware as a *protocol processor* device. Off-host protocol processor designs have been developed based on the use of a custom VLSI chipset, a general-purpose single-board computer, or a hybrid solution using both general-purpose microprocessor hardware and custom VLSI. This work is surveyed in detail in chapter three.

VLSI implementation is performed through the design of a chipset consisting of several application-specific integrated circuits. It has the advantage of (1) providing the very fast implementation of protocol functions through the use of custom circuits and (2) execution of protocol operations in parallel. The major drawback to this approach is that it is difficult to design VLSI circuits which perform complex tasks. As a result, the class of protocols that can be implemented in VLSI may be only those designed to be simple enough for that task [8][44].

With a general-purpose single-board computer (SBC), protocol operations may be carried out by a RISC or CISC microprocessor. Such an approach has the advantage that it is capable of handling even the most complex of protocols using software similar to that of an on-host implementation. The microprocessor approach has several drawbacks, however, with respect to performance. The speed of particular protocol operations performed on the microprocessor will almost certainly be lower than that of such operations performed with

specialized VLSI circuits. In addition, the parallelism between individual protocol operations obtainable with VLSI circuits can simply not be achieved with a single microprocessor. The SBC approach may also suffer from fact that most general-purpose SBCs are not likely to feature an adequate combination of stock components, such as a large bank of high-speed static RAM, DMA controllers, and an on-board network interface, which may be necessary for a high-performance protocol implementation.

Hybrid solutions may be composed of a combination of custom VLSI circuitry and perhaps several microprocessors. With clever design choices, it may be possible to achieve much of the high performance and parallelism of a pure VLSI approach while still being able to tractably implement complex network and transport layer protocols by taking advantage of the flexibility of the SBC approach.

1.4. Possible Advantages of an Off-Host Protocol Implementation

Regardless of the specifics of hardware-assisted protocol implementation, the approach offers several potential advantages to network applications. An off-host communications architecture can benefit both the host processor and the protocol implementation. As a result, host applications can run faster and more predictably and the performance of the protocol services provided to such applications can be optimized.

1.4.1. Host Benefits

When using an off-host communications architecture, the host processor can potentially receive the following advantages:

Reduced host load. Since all protocol processing activities are handled on the protocol processor, no host processor cycles are consumed for protocol processing.

Predictable application processing. Since transport and network protocols perform retransmissions, react to incoming traffic, and sometimes block on network access, the execution times for protocol processing are nondeterministic. When using an off-host architecture, such nondeterministic activity occurs on the protocol processor and not on the host.

As a result, the CPU demands of host processes become more deterministic, thus promoting the predictable operation of such processes.

Reduced and bounded host interrupt arrivals. A transport or network layer protocol typically receives interrupts from its underlying data link service in order to indicate frame-oriented activity. Some minimum number of frames are necessary for each transport service data unit (TSDU), and, in addition, an unbounded number of frames may arrive from other hosts or result from retransmissions. Each frame can cause a processor interrupt. When transport and network protocols run on a protocol processor, the host can be shielded from these interruptions. In such a scenario, host interrupts need only be performed by the protocol processor; thus, they can occur on a per-TSDU, rather than a per-frame, basis. This may result in substantially fewer host interrupts per TSDU. At the very least, these characteristics enable application processes to place a bound on the amount of interrupt handling in which the host will engage.

Reduction of incident TSDU traffic. One feature of hardware implementations of data link protocols in broadcast local area networks is that they, rather than the host CPU, filter incident traffic that is not addressed the host. Similarly, a transport and network protocol processor can deliver to the host only those TSDUs which the host desires to receive, thus shielding it from an accidental or malicious barrage of TSDU arrivals. Such a feature has been referred to as a "network firewall" [27].

1.4.2. Protocol Benefits

A communications protocol can receive the following benefits when run on protocol processor hardware:

Dedicated processing cycles. When run on the host, a communications protocol must contend with application processes for processor cycles, thereby losing potential CPU cycles and incurring overhead from scheduling and context swaps. These effects can reduce the amount of work the host processor can perform for its application, cause slow response

to protocol events, and lower protocol throughput. With dedicated protocol processing hardware, all processing cycles are available to the protocol, thereby raising throughput, and context swaps can be avoided, thereby lowering latency and preventing protocol buffer overflow. Furthermore, such dedicated cycles, may be crucial in order for the protocol to handle the real-time processing requirements of continuous-media traffic such as voice and video.

Specialized hardware. Network and transport protocols can benefit from special hardware such as high-speed RAM, efficient DMA controllers, and custom checksumming circuitry. Such hardware may allow certain protocol operations to be accelerated or performed in parallel. Components providing these services can be utilized on the protocol processor device, providing performance not attainable with the general-purpose hardware of the host system.

Optimized data path to the network interface. Modern transport and network layer protocols make multiple accesses to the network interface for each TSDU passing through the session layer. Due to segmentation, protocol headers and trailers, and retransmissions, the traffic across the boundary between the data link layer protocol and the network layer protocol can be much heavier than that across the boundary between the session layer protocol and transport layer protocol. Thus the throughput of the former pathway should be optimized. One method of optimizing this path is through the placement of the network interface on the same circuit board as the protocol processor device.

Ideal operating system environment. It has been recognized that an operating system environment can place severe restrictions on a protocol's performance and proper implementation [48]. The timer, buffer, and lightweight process management services provided by the host operating system may be relatively inefficient for protocol tasks. In addition, the host operating system may impose security restrictions and other overhead which can hinder protocol performance. With an attached protocol processor board, one is free to choose the ideal operating system for the protocol implementation, including none at all.

1.5. Possible Pitfalls of Using an Off-Host Protocol Implementation

Several pitfalls may be encountered when using an off-host protocol implementation. The first set of pitfalls concerning operating system overhead are shared by both on-host and off-host approaches; however, the other pitfalls, those of overtaxing the host CPU and complicating the data path, are unique to off-host architectures.

1.5.1. Incurring Operating System Overhead

It is fallacious to believe that the problems of operating system overhead discussed in section 1.3.1 no longer exist when one uses an off-host protocol architecture. In secure, multitasking operating system environments the protocol processor hardware becomes a shared resource which must be managed by a trusted server process or the kernel. As a result, much or all of the system call and context swap overhead experienced in-host implementations can still occur in the off-host case. Address space complications imposed by the operating system may also pose a problem.

1.5.2. Overtaxing the Host Processor

The benefits of off-loading the host by partitioning protocol processing onto an attached processor may be eroded due to several sources of overhead:

Control, Status, and Data Transfer. When using an off-host architecture, the host processor has to perform a certain amount of communication with the protocol processor device. The host processor must, to some extent, drive the protocol processor by sending it control information and reading its status. Such overhead can not be avoided; however, care should be taken to minimize it, lest the host processor become overburdened. The transfer of TSDUs to and from the protocol processor is a prime concern. If the host processor is burdened with this task, the data transfer may consume a significant number of processor cycles. This overhead may be amplified if the protocol processor resides on a separate board on the host's I/O bus. This bus may have relatively low bandwidth and high latency in comparison to the data paths on the CPU board. The designers of an off-host protocol

architecture should seek to minimize the bus transfers required by the host CPU to use the services of the communication processor.

Management of Shared Data Structures. The host may also be made to allocate shared data structures such as buffers for communication between itself and the protocol processor. Similarly, designers may wish for the host to manage various protocol processor resources such as open connections. These tasks should ideally be performed only by the protocol processor device. If not, they should at least be avoided in the common case of data transfer.

Multiplexing and Demultiplexing Command Requests and Status Responses. In situations where multiple application processes may use the protocol processor, the host may also be required to multiplex protocol commands descending from multiple processes and demultiplex protocol status ascending toward one of many processes. This may require the protocol processor device driver to manage a state vector mapping protocol processor response data to a particular process. The management of such a data structure results in another source of host processing overhead.

Notification of Command Completion. Since the protocol processor executes asynchronously with the host processor, there must be a way for the host to discover when protocol processor status responses have arrived for it to receive. This results in the need for either an asynchronous notification mechanism such as hardware interrupts or the use of periodic polling of the protocol processor response stream. The host must expend some processor cycles to perform these tasks and also to act upon the such responses when they are present.

1.5.3. Increased Data Path Complexity

Another set of pitfalls results from the complexity introduced by the addition of the protocol processing hardware into the network data path. These pitfalls may significantly affect protocol performance.

Low Bandwidth Connection to the Protocol Processor. The protocol processor may communicate with the host processor and host memory over a data path having relatively low performance. For example, the protocol processor may reside on a separate circuit board which resides on the host computer's I/O bus. Such a bus typically has lower throughput and higher latency than the host's CPU-memory bus. As a result, the off-host architecture may cause a performance drop on accesses to memory. Similarly the off-host architecture may force the data path between the protocol and the network interface to be over a low-performance medium; this may further degrade performance.

Additional Data Copies. Data from the higher-layer protocols may reside in the host's on-board memory, and addressing restrictions may require that this data be copied to local memory on the protocol processor device for processing. This copy occurs in addition to one that must eventually be performed to the network interface device. The effect of this additional copy may be a profound increase in latency. Furthermore, if the two copies are not performed in parallel, they may also have a serious effect upon protocol throughput.

Additional Bus Contention. If the protocol processor resides on the host's I/O bus, communication between it and the host may result in contention with other bus mastering devices such the network interface. This contention may erode system parallelism or may cause unpredictable protocol throughput or response times. Even if simultaneous requests for the use of the bus are not present, there may be an increased cost in the overhead of bus arbitration time due to alternating requests for bus mastership.

1.6. Summary

In this chapter we have motivated the study of off-host communications protocol architectures and discussed the possible benefits and pitfalls of employing such designs. In chapter two we cover the design issues inherent in off-host protocol architectures. Related work on off-host communications architectures from the literature is summarized in chapter three. Chapter four presents the design and implementation of an actual off-host com-

munications architecture, and chapter five summarizes and profiles its performance, identifying system bottlenecks. In chapter six we develop an analytic model to predict the performance of off-host communications architectures, apply it to our own design, and validate its predictions against the observations in chapter five. In chapter seven we conclude, evaluating our off-host communications architecture and analytic model and making suggestions for future work. Methods of obtaining input parameters for the analytic model are presented in the appendices.

2 Design Issues in Off-Host Communications Architecture

2.1. Overview

In this chapter we present the issues intrinsic to the design of off-host communications protocol architectures. These center around the components of such architectures, their placement, and the communication between them. We first consider the interface between the host and protocol processor. Later we discuss the architectural levels spanned by an off-host communications architecture, followed by the hardware and software mechanisms used to communicate between the components at each level. In conclusion we discuss the philosophical extremes of off-host communications architecture design.

2.2. The Host-Processor Interface to the Protocol Processor

We restrict our discussion to communications protocol architectures in which the transport and network layer protocols run on an off-host protocol processor, the higher-layer protocols reside in the host, and the data link layer and below are on a network interface device[1]. Given this situation, we must have the ability to send and receive a transport service data units between the host CPU and the protocol processor. In addition, we must have a way to command transport layer services from the protocol processor and receive an indication of protocol processor status. Some architectures may employ additional mechanisms to provide options with the command and status information. In this section we discuss the attributes of the various streams over which this information must flow. These streams may span several architectural layers, and several communications mechanisms may be employed to implement their interlayer flow of information. These issues will be covered below in sections 2.3 and 2.4.

1. The issues involved in the protocol processor's communication with the network interface are similar to those of the host CPU's communication with the protocol processor. Furthermore, the network interface may be integrated into the protocol processor, eliminating these communication issues altogether. Therefore such issues will not be covered in this work.

2.2.1. Command Stream

The session layer must be able to request various services from the transport layer. When using an off-host protocol implementation, the session layer must request these services through the use of a well-defined set of protocol processor commands issued by the session layer to the protocol processor device driver, which is charged with submitting them to the transport layer on the protocol processor hardware via the *command stream*. As a result, some host processing overhead for executing protocol processor device driver software is unavoidable when using an off-host communications architecture. Since the protocol processor executes asynchronously, there must be a mechanism for the protocol processor to recognize that a command is pending for it to process. This may be accomplished through hardware interrupts or through polling as discussed in section 2.4. An additional concern is that the interarrival time of commands from the host CPU may be shorter than the protocol processor service time for such commands. As a result, queuing of protocol processor commands may be necessary, and a stochastic queueing delay may become a component of the overall latency of the off-host communications architecture. The queue may utilize one of several queueing disciplines such as first-in-first-out or priority-driven. In addition, such a queue is likely to be of finite length, or there may be a limit to the number of commands which can be pending for a given transport service access point. As a result, the session layer may have to block on submitting a command to the protocol processor device.

2.2.2. Status Stream

The protocol processor device driver must somehow become aware of the status of commands submitted to the protocol processor. This can be done using two styles of interaction and a *status stream*. One form of interaction is that the session layer may query the status of a command issued to the protocol processor. Another is that the transport layer may indicate to the session layer asynchronously that an event has occurred. These styles

of interaction are similar to the use of interrupts and periodic polling. The protocol processor and its device driver may interrupt the session layer to indicate asynchronously the presence of pending status information from the transport layer, or the session layer may poll to synchronously read the status of the transport protocol as provided by the protocol processor. Interrupts and polling are discussed further in section 2.4. As with the command stream, the interarrival time of status information from the protocol processor may be shorter than the time for the host CPU to consume such information. As a result, queuing of protocol processor status may be necessary, and another source of stochastic queueing delay may be introduced into the aggregate latency of the off-host communications architecture. As is the case with a command queue, a status queue may employ one of many queuing disciplines. Note also that if there is a limit on the length of the queue used to supply status to the host CPU, the protocol processor may be forced to block on providing status to the host.

2.2.3. Data Stream

Each command or item of status information passed to or from the protocol processor may have a Transport Service Data Unit (TSDU) associated with it. This TSDU must be carried over some *data stream* between the two processors. It is important to point out that this data stream need only logically carry the data; that is, the data need not be physically copied between the protocol processor and host. In fact, copies of protocol data should be avoided since significant latencies and loss of throughput for large messages can result from introduction of extra data copies in a communications architecture. Architectural considerations and the nature of particular transport and network layer protocols may conceivably dictate the need for a physical data copy, however. Note that all data must undergo at least one physical copy in the communications architecture; such a copy is either to or from the network interface hardware. The data may pass either physically or logically through the data stream between the host and protocol processor, but it must always move physi-

cally through the network interface. In contrast with the command and status streams, queueing of data in the data stream may not be physically necessary since pointers to such data may be part of the command or status information which already undergo any necessary queuing. If queueing is required, however, it introduces issues similar to those discussed above.

2.2.4. Options Stream

An *options stream*, similar to the data stream, may also be employed to pass options associated with commands and status between the host and protocol processor. Commands which require these options will incur additional overhead on both the host and protocol processor since additional processing is required to transfer and process this information. The benefit of an options stream is that it can minimize the amount of information present in the command and status streams. Information that need not flow in the common case of protocol processor activity can be left out of the command and status stream and included in the seldom-used options stream. The end result is that a minimum amount of command and status traffic occurs in the common case.

2.3. Hardware and Software Architectural Levels

An off-host communications architecture consists of several levels of software and hardware architecture. These levels are the user level, kernel level, bus level, protocol-processor level, and network-interface level; the user, kernel, protocol-processor, and network-interface levels are structured such that they lay atop one another. The user and kernel levels are simply subdivisions of host software, whereas the protocol-processor and network-interface are hardware levels separated from each other and the host levels by the bus level. The relationship between the levels is illustrated in Figure 2.1. In some architectures, the kernel level is not present, or the protocol-processor and network-interface levels are unified. As discussed below, the nature of these levels is somewhat dictated by the operating system environment and hardware organization of the host computer system.

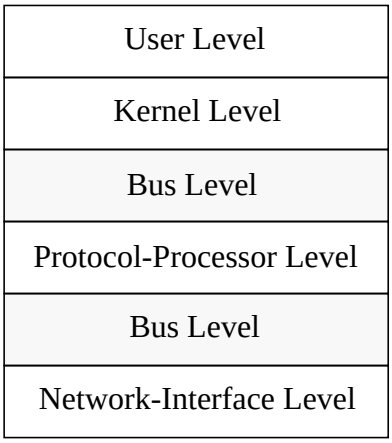


Figure 2.1: Architectural Levels

The components of the communications architecture, including the application program interface and specific protocol layers of the ISO OSI reference model, are distributed over each of the architectural levels. In this work we consider communications architectures in which the transport and network layer are implemented off-host on a dedicated protocol processor, the session layer and above are implemented in-host, and the data link layer and below are implemented on a network interface device. Our concern here is chiefly with the details of the protocol processor and its interconnection to other architectural components; however, we briefly sketch certain details of the implementation of surrounding protocol layers in order to explain how they fit into the complete off-host communications protocol architecture.

2.3.1. User Level

The user level refers to that at which application processes run in the host operating system. Such processes may send and receive application service data units (ASDUs) located in the memory of their address space. Applications transfer these ASDUs by invoking communication primitives from an Application Program Interface (API). An example of such an API is the socket interface of BSD UNIX [30]. At least part of the API imple-

mentation must reside at the user level so that it may be accessed by applications; however, much of the API implementation may also occur at the kernel level.

Another component of the communications architecture which may reside at the user level is the protocol processor device driver. For host operating systems such as MS-DOS [18] in which secure access to shared resources is not a concern, the protocol processor device driver may safely run at the user level. In such cases, the driver may be in the form of library code linked with an application program or it may be installed as a operating system device driver where its code still resides in the user-level address space.

In host computer systems with multiple processes or users, secure access to the protocol processor device may be a concern. In such cases it may be necessary to place the protocol processor device driver in a protected operating system kernel. An alternative to this approach, employed in operating systems such as Mach [1], allows device drivers to run securely at the user level. Such drivers run as privileged server processes as discussed in section 1.3.1. In these scenarios a small amount of kernel software may be required to perform privileged hardware operations that manage processor interrupts or to map hardware registers into the server process' address space. Note that in order to provide protection for shared resources, the server process' address space, albeit at the user level, must still be somewhat disjoint from that of the client processes requesting protocol processor service.

Program code for implementations of protocols at and above the session layer may also reside at the user layer. Such protocols may be in the form of library code linked with the code of an application, or the protocols may run as server processes.

2.3.2. Kernel Level

In operating systems such as UNIX [30] which provide the secure management of shared resources for multiple user processes, host memory is divided into protected address spaces. Each process has a private address space which it is separated from that of other processes and especially the operating system *kernel*[2]; the process may only access its own

address space. This protects kernel data structures and hardware devices from unsafe or malicious use. To ensure the secure use of an off-host protocol implementation in such an environment, the protocol processor device driver may be run in the operating system kernel. (A secure alternative employing a protocol server is discussed in the previous section.) When at the kernel level, the device driver executes in its own distinct address space. User-level applications may request secure services from the device driver through a restricted set of system calls. Software executing in the kernel layer typically executes in a privileged CPU mode that enables certain processor instructions useful in device drivers which are unavailable to user-layer software. For example, only kernel-level code may be able to execute privileged I/O machine instructions or enable, disable, and handle hardware interrupts. The software implementation of protocols at or above the session layer may also reside at this layer as may portions of the API implementation.

2.3.3. Bus Level

The hardware components of an off-host communications architecture must be connected with some variety of buses. Such hardware components are the host CPU, protocol processor, network interface, and memory. All communication between these components must occur over a bus as discussed in section 2.4.2.3. Note that there may be several buses within the host computer system and that some buses may allow multiple masters and some may not. The individual buses may have varying performance characteristics. One specialized type of bus is designed to have high performance, and is used to connect only a few high-speed components such as a processor and memory; such a bus is often termed a *memory bus*. Another type of bus is designed to be versatile; it typically connects I/O devices with varying response times and is termed an *I/O bus*. An I/O bus typically has slots to allow the insertion of several *modules* which are simply circuit boards that may contain

2. Note that in many operating systems, the term *kernel* refers only to the software which provides core system services and encapsulates the hardware; security features such as a protected address space and privileged instructions are not implied by this term. It should be recognized that in this work we use the term kernel in a stronger sense, implying the presence of these security features.

memory, a processor, or an I/O device. Some rare memory bus designs may also support the insertion of a small number of high-performance modules. Due to their flexible design, I/O buses typically have lower performance than memory buses.

2.3.4. Protocol-Processor Level

As discussed in section 1.3.2, the protocol processor may take the form of a custom VLSI application specific integrated circuit (ASIC) chipset, a general-purpose SBC, or a hybrid of the two. In this thesis we consider the case where the protocol processor implements both the transport layer protocol and the network layer protocol. Such a processor may reside either directly on the host CPU circuit board or as a module on a bus connected to it. The protocol processor may use its own local memory which may allow a faster access time than that of the memory used by the host processor. In fact, it may even be difficult or impossible for the protocol processor to address specific regions of host memory. This can be particularly true when the host uses virtual memory. The protocol processor makes use of a network interface device supporting the data link layer and below.

2.3.5. Network-Interface Level

In chapter one, we refer to the device supporting the data link layer and below as the network interface. This device is used by the protocol processor to transmit and receive data link layer SDUs, hereafter referred to as *frames*, to and from the network. The network interface may reside on the same board as the protocol processor, or it may be on a separate bus module. The network interface is unique in that it has some special data interface such as FIFO memory buffers which temporarily hold incoming and outgoing frames. It is at some point necessary for frames to be copied to these buffers in order to reach the network, regardless of the protocol or communications architecture in use. Similarly, it is necessary for incoming frames to be copied from these buffers. Hence, all communications architectures will require *at least one* copy of the data contained in application service data units.

2.4. Communication Mechanisms Between Levels

A critical issue determining the performance of a communications architecture is the set of communication mechanisms employed between its components. This section describes the possible mechanisms which may be used between the components at each layer of a communications architecture. We first examine communication between the software components of the host computer system, followed by that between the hardware components of the communications architecture.

Before we examine how data transfer can occur between the layers of the communications architecture, we first review what sort of information will be transferred. We take as an example the interlayer communication necessary for transmission of an ASDU onto the network media from an application process. The ASDU exists as a data structure physically located in the host computer's memory and logically in the user-level address space of the application process. It passes through the protocol suite enroute to the network. As it does, protocol control information is appended to the message as it becomes an SDU for the various protocol layers. This data migrates through each of the architectural levels discussed in the previous section. In addition to this data transfer, control, status, synchronization information, and perhaps options must pass between certain layers of the communications architecture. For example, a protocol processor device driver at the kernel level must write control information to the protocol-processor device at the protocol-processor level and read the device's status. In addition, the host CPU may need to synchronize with the protocol processor in order to notify it that a command is pending. Likewise, the protocol processor may need to notify the host of the completion of a command.

2.4.1. Communication Between the User and Kernel Levels

In off-host communications architectures where the protocol processor device driver resides in the host operating system's kernel, it is necessary to communicate between the kernel level and the user level. Even if the device driver resides in a protocol server, a

limited amount of communication with the kernel may still be necessary to perform privileged instructions. The mechanisms described below facilitate the exchange of information between the user and kernel levels.

2.4.1.1. System Calls

For off-host communications architectures in which the protocol processor device driver resides in the host operating system's kernel, a *system call* mechanism is likely to be necessary for communication between the user and kernel levels. System calls allow user-level programs to obtain operating system services. In the case of off-host protocol implementations, we are concerned with the control of the protocol processor device; hence, we shall limit our scope to the class of device management system calls. In our case, these calls are ultimately supported by the protocol processor device driver. We shall be most concerned with system calls which allow a user process to *write* information to a device or to *read* information from a device.

A system call provides a service similar to a subprogram call in that it invokes a specific software routine (in our case, a device driver routine); however, it provides this functionality in a unique way since the routine resides at the kernel layer. To ensure security, system calls only allow access to a restricted set of operating system routines at well-defined entry points. Furthermore, the system call mechanism switches the mode of the processor so that it may access the kernel-level address space. It is also quite likely that the system call will also place the processor in a privileged state in which it may access a larger set of machine instructions such as those which allow the manipulation of I/O devices.

A system call is further similar to a subprogram call in that it may pass a set of parameters and possibly return a value. However, the fact that the kernel may execute in a distinct address space complicates this metaphor. Like subprograms, parameters may be passed to system calls on the user process' stack or in its registers; this is analogous to the *call-by-value* parameter passing mechanism for procedures. If the kernel executes in a dif-

ferent address space, it may have to incur some stack or register mapping overhead to be able to manipulate these parameters. Similar efforts are likely when pointers to memory regions in the calling process' address space are passed to the kernel; this is analogous to the use of the *call-by-reference* parameter passing mechanism in procedure calls. In this case, the region referred to by the pointer must be made visible in the kernel-level address space. This issue is discussed in the next section.

Like a subprogram call, a system call requires a certain amount of overhead to modify, save, and restore the state of host CPU registers. However, the system call will require additional machine status to be saved and restored (possibly the entire process context). As a result, its overhead is typically much larger than that of a procedure call.

2.4.1.2. Shared User- and Kernel-Level Memory Regions

Recall that the user and kernel levels typically have protected address spaces. In some systems, an address-range or a page- or segment-table protection scheme may be used to prevent user processes from accessing memory allotted to other processes or the operating system kernel. Systems with virtual addressing typically protect private memory by providing user processes and the kernel with disjoint virtual address spaces. In either of these scenarios, a method of communication between these layers is to selectively remove this protection over certain memory regions, thus allowing them to be shared. First we shall review how such schemes may be achieved, and then we shall discuss their utility in an off-host communications architecture.

For host operating system environments in which an address-range protection or a page- or segment-table protection scheme is used, it is usually the case that user-level processes are prohibited from reading or writing the memory regions of other processes or the kernel. In some cases these restrictions may also apply to the kernel. In such systems, however, the kernel may modify the protection range over a region of kernel- or user-level memory or it may enable read or write protection on a range of page or segment table

entries. This allows unrestricted, shared access by the kernel-level or user-level entities to each other's memory.

In host systems with virtual addressing, user-level processes are typically given a disjoint virtual address space from other processes and the kernel. This prevents user-level and kernel-level entities from addressing each other's memory, and, therefore, sharing it. To remedy this situation, the kernel may access regions of the user-level address space by *mapping* them into its own address space. In systems employing a protocol server architecture it may be desirable for the privileged protocol server to map regions of the kernel-level address space into its user-level address space. Such mappings may also be employed between the server and its client processes. The protocol server may need to invoke the kernel in order to perform such mappings. An issue unique to virtual-addressing systems is that a pointer to an object passed between two virtual address spaces is not valid if interpreted in the foreign address space. Some means of translating such a foreign pointer (so that it refers to the local, mapped version of the object) must be employed for the mapped version to be accessible.

A similar use of mapping between a host virtual address space and a physical I/O address space may be required to allow a protocol processor in the I/O address space to address user-level or kernel level memory. A unique issue here is that, in host systems with virtual memory, it may be necessary to "lock down" pages of user-level memory such that they will be physically resident when accessed by a protocol processor in I/O address space. Communication with the protocol processor is discussed further in section 2.4.2.

The point of creating shared memory regions of user-level and kernel-level memory is to allow protocol data units to cross the barrier between the user and kernel level without being physically copied. The minimization of protocol data copies typically helps to ensure high performance. This is likely the case for large data units, where the overhead of the creation of shared memory for each message is low compared to passing the information by value on a system call parameter list. For short messages, however, a mechanism employ-

ing the call-by-value mechanism may prove to have higher performance since such a method avoids the overhead of shared memory creation.

2.4.1.3. User-Level Interrupts

Some operating systems such as UNIX [30] allow kernel software to interrupt a user process in order to notify it of the occurrence of an asynchronous event such as the arrival of protocol processor status. This information may be sent by a kernel-level device driver to an application process or a protocol server. Within the user level, a protocol server may also use such a mechanism to notify its clients of pending responses. These interrupts require processor context to be saved and possibly may change the mode of the processor. As a result, they can have a relatively high overhead. Further problems with interrupts are discussed below in section 2.4.2.6.

2.4.2. Communication Between Hardware Components

We assume an off-host communications architecture that consists of the following principal hardware components: a host CPU, a protocol processor, a network interface, and memory. This section is concerned with the communications mechanisms used between these components. The memory component is unique in that it may be distributed within each component or exist as a peer component. We shall focus primarily on communication between the host and protocol processor, although many of the mechanisms discussed may also be used to communicate with the network interface. Furthermore, we recognize that the protocol processor and network interface may be integrated into a single unit; in this case, communication between these components is a non-issue.

We first consider the high-level mechanisms which may be used by processors to convey information to other processors or devices. These mechanisms are shared memory and I/O registers. At a lower level, we consider the issues involved in physically transferring data over the bus connecting circuit boards on which the processors, memory, and devices reside. We then examine two approaches to driving the transfer of information over

the bus: programmed I/O and direct memory access. Finally we consider interrupts and periodic polling, methods of communication which simply act to synchronize the otherwise-asynchronous components of the architecture.

2.4.2.1. Shared Memory

One mechanism for communication between the host processor and protocol processor is the use of shared memory. With this method, a common region of memory is shared between the two processors. Once such a memory region is prepared, its naive use is quite straightforward; one processor simply writes information to this memory for the other to read and vice versa.

This naive approach is not adequate, however, because the two processors accessing the memory execute asynchronously. As a result, some form of reader/writer synchronization is required, lest the reader read invalid data or the writer overwrite valid data. Various synchronization protocols may be used as discussed in [22][29][40]. The protocol processor and the host must also agree on where items in the shared memory are located, and they may also need to be able to allocate subregions of the memory for specific uses. One approach to ensuring that no two memory subregions allocated by the host or protocol processor overlap due to a lack of processor synchronization is to charge one processor with the allocation duties. Ideally the protocol processor shall be charged with most of the synchronization protocol tasks in order to off-load the host (a major goal of off-host protocol processing); however, in many cases it may be more straightforward to have the host perform these duties.

In addition, the very feasibility of the creation of a shared memory region is an issue. For both the host CPU and protocol processor to be able to access shared memory they must both be able to address it. Thus, to create a region of shared memory, there must be a region of memory common to the *address spaces* of both processors. In some computer systems, certain memories may not have this property. In others, such a region may exist,

but it may be quite small and may be divided for use with other I/O processing devices. As a result, there may be severe restrictions on its use, including constraints on the size and location of the subregions that may be used.

The access time of the shared memory in use by the host and protocol processor may have a significant effect on the performance of the off-host protocol processing architecture. Special high-speed memory may be employed on the protocol processor device in order to optimize protocol processing and achieve higher performance. It could therefore be a good design choice to use this high-speed memory as the shared memory region if it is mutually addressable. However, if the use of such memory results in additional copies of protocol data units, the benefits of its short access times may be lost, and lower performance may result.

2.4.2.2. I/O Registers

An alternative to the use of shared memory for communication between the protocol processor and the host CPU is the use of I/O registers on the protocol processor device. I/O registers are addressed by the host in either its physical memory address space or a distinct I/O address space depending on whether the CPU supports memory-mapped I/O or isolated I/O. These registers are command registers, status registers, and data registers. Some registers have access restrictions; a CPU may only write to command registers, and it may only read from status registers. Data registers may be read-only, write-only or unrestricted. The host's device driver would use command registers to send commands and options to the protocol processor device indicating that it perform one of a well-defined set of transport protocol tasks. Later, the driver may query a status register to determine the status of the device or the commands issued to it. Data registers (likely FIFOs) may be used to transfer protocol data to and from the device. I/O registers may have relatively fast access times relative to shared memory; however, they may also have strange address alignment and data size restrictions which complicate their access. Due to their simple nature, I/O

devices without a microprocessor often use I/O registers rather than shared memory for communication with their driver.

2.4.2.3. Bus Transactions

Whether they use shared memory or I/O registers, separate components connected via a bus communicate with one another via bus transactions such as reads and writes. Specific components may act in bus transactions as masters, slaves, or both. A component behaving as a bus *master* is an active entity which may initiate bus transactions which transfer data to and from slaves on the bus. A component in the role of a bus *slave* is a passive entity which typically may only respond to requests from a master or generate hardware interrupts. Some components on the bus are capable of handling interrupts. Hardware interrupts are discussed further below in section 2.4.2.6. For buses on which more than one module may become a master, a medium access control scheme is required to prevent two masters from driving the bus at once. Access control is done through an arbitration scheme which chooses from possibly several requests originating from different bus masters and then grants control of the bus to one specific master. One of many arbitration policies (e.g., fixed priority or round-robin) may be adopted. An advantage to having only one possible master on a bus is that arbitration becomes unnecessary.

A processor (either the host CPU or the protocol processor) performs the transfer of protocol data units or control information over the bus through the use of multiple bus transactions. To perform a bus transaction, the master must have exclusive access to the bus. Thus, with multiple master buses, a master wishing to utilize the bus must first request the bus. This request may have to undergo arbitration with the pending requests of other requesting masters. Arbitration results in overhead in addition to that of the transaction itself. The choice of a bus *release* policy is another issue. A master may adopt a *release-when-done* (RWD) policy in which exclusive access to the bus is relinquished at the end of each transaction or group of transactions; otherwise, it may adopt a more greedy *release-*

on-request (ROR) policy in which exclusive access to the bus is held until other masters have pending requests. Bus arbitration overhead may therefore be minimized if the master employs a ROR policy.

Another source of overhead that may be particularly significant for bus *read* transactions is that the address of data to be read must initially be asserted on the bus by the master; after a time interval, data asserted by the responding bus slave becomes available for reading. For *writes* the master may assert the address and data simultaneously on the bus. As a result, bus writes often have lower latency than bus reads, and the off-host protocol processor can contrive to take advantage of this fact.

As another solution, two techniques may be used to improve the performance of reads. For some buses, read transactions may be *pipelined* such that while the data from the previous transaction is being asserted by a slave, the address for the next read transaction can be asserted by the master. This parallelism removes some of the overhead of the read transactions. Another technique which may be used in transfers of contiguous memory is the use of *block mode* transactions. With such a transaction, the master asserts only the starting address of the memory to be transferred; the data are asserted without any further addressing overhead while the slave reads it from or writes it to successive memory words or a FIFO register. In addition to lowering the overhead of read transactions, this scheme may also be used to improve the performance of writes.

In order to facilitate system expansion and support a wide class of I/O devices, backplane buses are typically designed to have many slots and support various device speeds. As a result, electrical considerations usually require the bandwidth of a backplane bus to be lower than that of a memory bus.

In addition to the problems of arbitration overhead in multiple-master buses, contention for the bus can force requesting masters to wait for the transactions of other masters to complete. Various arbitration policies may resolve this contention in different ways including preemptive policies which cause the wait times of high-priority requests to be

short; however, significant aggregate delays can result which may hurt the overall latency of the communications architecture. Even more significant may be the effect of bus contention on overall throughput. If the protocol processor and the host CPU share a common bus it may remove the possibility of performing data transmission in parallel. As a result, copies that would otherwise occur in parallel may degenerate into serial copies, and aggregate throughput may suffer greatly as a result.

2.4.2.4. Programmed I/O Transfers

When data is transferred between components with a multiple-master bus, a distinct issue is determining which entity drives the transfer. We refer to a transfer in which a CPU performs the data copy as a *programmed I/O* transfer. In an off-host communications architecture this CPU may be the host CPU or that of a microprocessor-based protocol processor. In such a transfer, the memory movement or I/O instructions of the CPU are used to read or write data over the bus under program control. Programmed I/O transfers are quite straightforward in that they do not inherently require any set up or synchronization between the endpoints of the transfer. The major drawback to programmed I/O is that it consumes CPU cycles which may be put to better uses. Furthermore, the efficiency of burst mode bus transactions may be unavailable when using programmed I/O. It is clear that the use of programmed I/O by the host CPU in an off-host communications architecture erodes the benefit of the architecture with respect to off-loading the host.

2.4.2.5. Direct Memory Access Transfers

An alternative to programmed I/O for the transfer of contiguous memory regions is the use of *direct memory access* (DMA) transfers. With a DMA transfer, a processor programs a special circuit known as a *DMA controller* to perform the data copy. A CPU must prepare the controller for the transfer by providing it with the base addresses of source and target memory buffers involved in the copy (one of which may be implicit) and the length of the transfer. Then the processor commands the DMA controller to start the transfer. The

chief advantage of DMA transfers is that the processor does not drive the copy and is therefore free to continue processing in parallel. One drawback of the use of DMA transfers is the overhead of setting up the transfer. Another is that the DMA transfer is performed asynchronously with the processor, and as a result some synchronization overhead must be incurred in order to inform the processor of the completion of a DMA transfer. Thus, a processor may spend more cycles using DMA for short buffers than if it had simply employed programmed I/O.

2.4.2.6. Hardware Interrupts

Note that the host CPU, protocol processor, and network interface hardware are three autonomous components in the host computer system. Each executes its own set of tasks asynchronously with respect to the other components. A wholly different problem from that of transferring data between the components of an off-host communications architecture is that of synchronizing these hardware components. *Hardware interrupts* are one mechanism with which to perform this task. A processor or device may asynchronously inform a processor of an event by sending it a hardware interrupt. This interrupt causes the processor to postpone further processing of instructions in order to execute an *interrupt service routine* (ISR) containing software which has been set up to handle the event. For operating systems which provide security between multiple processes, interrupts often change the processor mode of the host to provide it with kernel privileges or change its address space. This may allow the processor to perform privileged operations necessary to handle the interrupt. The interrupt mechanism is advantageous in that it provides a timely response to the event and incurs overhead only when events occur.

Hardware interrupts may also introduce a number of problems. Since they require a processor to postpone the further processing of its instructions and return later, the processor context must be saved before the ISR begins to run, and it must be restored afterward, otherwise unpredictable operation would result. The overhead of this context switch

may be significant, especially if the processor mode, address protection, or virtual address space must be changed. Another issue is that since interrupts postpone the execution of processor instructions, they change the timing behavior of the code and thereby detract from the predictability of real-time processing tasks. Furthermore, if the processor instructions interrupted and the ISR routine invoked use shared resources or data structures, some form of concurrency control may be required in order to avoid software bugs. To combat these problems, interrupts may often be prioritized and the mode of the processor may be set such that interrupts below a certain priority level are postponed or ignored. The use of this feature also allows pending interrupts of higher priority to be serviced before those of lower priority. The drawback to disabling interrupts at a particular priority level is that it delays the response to the event which generated the interrupt.

2.4.2.7. Periodic Polling

An alternative to the use of interrupts is *periodic polling*. With this mechanism, a processor periodically checks for the occurrence of asynchronous events. These events may be indicated in status registers on a device or they may be encoded in data structures stored in shared memory between the processor and the device. The major drawback to periodic polling is that it absorbs host cycles even when there are no pending events. This approach is attractive, however, in that the problems inherent in interrupting processor instructions are avoided. Particularly, the predictability of real-time processing may be much easier to ensure with a polling approach. Furthermore, with polling, the use of shared data structures by event handling code and other software is readily serialized.

2.5. Architectural Integration Continuum

The previous sections introduced a wide variety of design options for the use of architectural levels and methods of communication between those levels. In this section we present the two extremes of a continuum of design choices aimed at pursuing the benefits

of an off-host protocol processing architecture. We also discuss the trade-offs inherent in venturing toward each end of the continuum.

2.5.1. Continuum Choices

There are two fundamental design directions open to the developers of an off-host communications architecture in their attempt to provide the benefits of off-host protocol processing to host applications. Developers may choose a tightly-integrated design or a loosely-integrated one. In a *tightly-integrated* design, there are a minimum of software and hardware architectural levels and the data path between them is of minimal complexity. At the extreme, there is no kernel level and the architecture supports only independent, single-master buses. There is only one memory and it is shared symmetrically among all hardware components and resides in a common address space. The protocol processor and the network interface are integrated into a single unit and each is a custom ASIC. Finally, all components are connected via direct bus pathways and some addressing logic.

In a *loosely-integrated* design, all hardware and software layers are present, the host CPU, protocol processor, and network interface are each on a separate circuit board residing on a multiple-master bus, and memory is distributed asymmetrically on several boards. There are separate user-level, kernel-level, and I/O address spaces. In addition, the protocol processor is constructed using a single microprocessor.

2.5.2. Trade-Offs

There are a number of trade-offs which the designers of an off-host communications architecture make in moving toward a tightly-integrated or a loosely-integrated extreme. These trade-offs may represent factors such as the development or production cost of the architecture, the modularity of its design, and its overall performance.

Since each of its components is on a separate circuit board, a loosely-integrated design may be fashioned from widely-available products. As a result the development costs may be relatively low; however, the production costs may be high since there are many

components. A tightly-integrated design requires the development of custom circuitry which may be an expensive development endeavor; however, the production costs may be low since a minimum of circuitry can be employed. A tightly-integrated architecture suffers from the drawback that it may only function with one type of host, one set of transport and network protocols, and with one data link layer protocol, physical layer protocol and network medium. In contrast, a loosely-integrated approach is modular; it may be integrated into a variety of configurations by substituting certain components. A loosely-integrated approach may require the host to have a common type of I/O bus into which a variety of compatible modules may be inserted. Due to the fact that it has a more straightforward data path than one designed from a loosely-integrated approach, a tightly-integrated design is likely to result in lower protocol latency. In a loosely-integrated design, the host CPU, protocol processor, and network interface are on separate circuit boards so they are likely to contend for the single I/O bus. This would hinder the parallel data flow among them. A tightly-integrated approach with multiple, single-master buses and a protocol processor and network interface on the same circuit board is likely to avoid bus contention; hence, it is more likely to achieve a high degree of parallelism than a loosely-integrated approach. This should yield relatively higher throughput.

The dilemma of the system architect is that all these design choices interact with one another such that no single set of choices guarantees an optimal design, i.e., there are trade-offs. The remainder of this thesis illustrates the subtle interactions encountered in various approaches to off-host protocol processing.

3 Related Work

3.1. Overview

Several efforts to develop communications protocol architectures that place the transport and network layer protocols off-host have been documented in the literature. This chapter surveys the related work, paying close attention to how each resolves the design issues presented in chapter two. We examine how each implementation is structured in terms of its user, kernel, bus, protocol processor, and network interface, and we observe the communications mechanisms between such components. Protocols present at and above the session layer are ignored.

In an attempt to evaluate architectural design choices, we also quote the performance results or estimates for each architecture when they are available. Some communications architectures resulted in the production of a prototype; in these cases, empirical performance results are available. Other projects only went through initial design and unit testing phases; such work provides only estimates of performance.

Off-host communications architectures have many similarities to parallel protocol implementations. To this end we wish to recognize the fine body of work on parallel implementations of network and transport protocols [6][7][16][17][20][23][25][49][50][51][52]. This research is not surveyed here because it is largely concerned with issues intrinsic to parallel host computers rather than with the off-host design issues presented in chapter two.

3.2. Chesson

A pioneering effort in the development of off-host protocol architectures was Greg Chesson's Protocol Engine project [9]. The goal of this project was to develop the Protocol Engine, a custom VLSI protocol processor for the Xpress Transfer Protocol (XTP). XTP possesses two unique architectural features relevant to off-host protocol processing. First,

it is a transfer layer[1] protocol spanning both the transport and network layers of the ISO OSI reference model. Second, XTP was designed as part of the Protocol Engine project to readily facilitate its VLSI implementation.

The design of the protocol processor is discussed in [10] and [44]. The most distinguishing feature of the Protocol Engine is its VLSI-intensive approach; in choosing an ASIC chipset implementation for the common-case processing of XTP, the Protocol Engine designers hoped to maximize the parallelism possible in protocol processing. For example, they designed the engine to overlap such operations as header parsing and address processing, and to pipeline the processing of back-to-back XTP TPDUs. Furthermore, the engine architects planned to take advantage of the high speed of custom VLSI circuits in order to process incoming MAC frames at the full bandwidth of a 100 Mbps local area network.

Several custom RISC sub-processors (engines), some with support for fast context switching, act in concert to support the parallel and efficient processing of protocol tasks. At the heart of the Protocol Engine are separate transmit and receive engines which handle protocol-specific processing for the outgoing and incoming TPDU streams, respectively. Checksums are calculated in hardware as TPDUs stream to and from the network. The other processors in the Protocol Engine are less protocol-specific. A special buffer-controller processor manages data buffers stored in two banks of DRAM, interleaved for fast access. In order to communicate with a variety of network interfaces (e.g., Ethernet or FDDI), the protocol processor uses a generalized MAC port which connects the protocol processor to the bit stream of an on-board network interface. A more general-purpose control processor handles complex XTP processing tasks such as network level route management and connection setup. Such are operations that do not occur in the common case of protocol processing.

1. The term *transfer layer* originates with the protocol hierarchy of the GAM-T-103 reference model [12].

To communicate with the host, the Protocol Engine includes a host port. Its circuitry transfers data buffers between host and protocol processor memory using high-speed DMA (its transfer rate was projected to be 200 Mbps). A host port connects the Protocol Engine to the host computer via the host's system bus, which may be one of various I/O backplanes such as the VMEbus or Sbus. The literature provides no details of the command and status streams between the protocol processor and a user-level application.

3.3. Kanakia and Cheriton

Convinced that conventional transport protocols were too slow without hardware implementation, Kanakia and Cheriton designed the VMP[2] Network Adapter Board (NAB) [27]. The NAB is an off-host protocol processor board designed to implement Cheriton's VMTP transport protocol [8]. In their design, VMTP runs directly above the raw data link, i.e., the network layer of their protocol architecture is null. Like XTP, VMTP is a simplified, lightweight protocol, designed to allow VLSI support.

The primary goal of the NAB prototype is to allow high speed operation of VMTP over an underlying 100 Mbps network. This is to be accomplished on the protocol processor through the use of custom hardware consisting of five major components, all functioning in parallel: an on-board network interface, a TPDU pipeline, a buffer memory, a host block copier, and a general-purpose, on-board processor. The block copier is essentially a DMA controller and is used for transferring commands, status, and data between NAB memory and host memory. The TPDU pipeline has the effect of hiding the overhead of certain protocol-specific processing operations (e.g., checksumming and encryption) for each 32-bit data word passing through the network interface. The pipeline is designed to operate at the full network data rate. It should be noted that such pipelining mandates the fixed-field encoding of VMTP TPDUs.

---

2. VMP is the name of the multiprocessor computer with which the NAB was to be used. The NAB design, however, is general enough to be used with other host computers as well.

Kanakia and Cheriton felt that the memory architecture performance of networking hardware was an issue of increasing importance, citing future gains in bus and network bandwidth. They also wanted a design that would not prohibit concurrent access from the parallel components of the NAB. With these issues in mind, they gave the NAB a novel memory architecture consisting of dual-ported, high-speed, static-column RAM (40-ns VRAM). One port is optimized for high-bandwidth (800 Mbps) sequential access during transfers by the block copier, and the other is made to support random access by the on-board processor. To avoid contention, these ports may be accessed independently. A dual-port DRAM controller chip arbitrates the use of the RAM by the TPDU pipeline, the host's VME system bus, and the on-board processor.

The on-board processor is a 16-Mhz Motorola MC68020. It orchestrates the operation of the buffer memory, block copier, and TPDU pipeline and performs a small amount of complex protocol processing required in the case of error-free protocol operation. In addition, the queueing of command and status information is handled by the on-board processor. All "special-case" protocol processing such as handling acknowledgments and retransmissions, is relegated to the host. Thus, this communications architecture is not purely off-host.

In addition to the goal of fast protocol processing, the NAB designers stressed the importance of off-loading the multiprocessor host by removing overhead in the form of host CPU cycles, system bus bandwidth, and host interrupts. The designers felt that the register-oriented RISC cycles of their host's high-performance CPU would be poorly utilized if burdened with the memory-intensive task of protocol processing. In addition, they desired minimal transfer of protocol data over the system bus since it is a critical multiprocessor resource, connecting the VMP processor nodes. Finally the VMP architects felt that without a NAB, excessive host interrupts from the high-speed network interface for each data link frame would result in poor cache performance due to the cache invalidations that must occur in order to swap in the interrupt handler context. Therefore they designed the

NAB to issue a minimum of host interrupts and to handle all interrupts from the network interface, providing a “network firewall.” Such a firewall would be particularly useful if an accidental or malicious barrage of traffic were to arrive at the host.

The NAB literature also details some of the issues of the command and status streams to and from the protocol processor. Command blocks are sent from the host to the NAB via a 1024-byte command and status register. Two such blocks are described in the literature; a Transmit Authorization Record (TAR) block commands the protocol processor to transmit a TSDU from host buffer space, and a Receive Authorization Record (RAR) block commands the processor to receive a TSDU into host buffer space. After being written to the I/O register, each command is copied by the NAB into a queue in its buffer memory. The data stream may be carried in one of two ways. Short TSDUs are embedded inside the blocks; long TSDUs are memory-mapped and referenced by pointers in the blocks. Without the authorization of a RAR, incoming TSDUs may not enter or interrupt the host; this helps to provide the firewall effect. After the operation of a command block completes, the NAB leaves status information in the status register and interrupts the host.

The literature provides performance estimates for the architecture derived from various architectural parameters such as the system bus transfer rate and the TSDU buffering latency. These estimates predict NAB throughput to be 44.3 Mbps for a 16 Kbyte transfer. The time between the start of a one-byte transmission request and the status indication of its completion is estimated at 1.6 ms. In the analysis, most of this latency is dominated by host processing time. Unfortunately, Kanakia and Cheriton provide no details on what specific host processing contributes to such latency.

3.4. Cooper et al.

Cooper et al. developed the Nectar Communications Accelerator Board (CAB), a flexible communications architecture in which various transport and network layer protocols may execute off-host on a general-purpose RISC processor, supported by some proto-

col-independent custom hardware [11][43]. This protocol processor is but one component in the Nectar project, which aims to construct a local area network that can function as a multiprocessor composed of workstation nodes. In this scenario, the designer's primary goal was to achieve low host-to-host message latency. The main benefit of employing a general-purpose processor rather than custom VLSI to execute protocols is that a variety of protocols may be implemented in software on the CAB, regardless of their complexity. To illustrate this, the CAB architects developed support for several network and transport protocols, including the ubiquitous TCP, UDP and IP protocols, as well as their own Nectar-specific Reliable Message Protocol (RMP).

For the system configuration discussed in the literature, the CAB resides on the VME backplane of a Sun-4 host running an unspecified flavor of UNIX. The protocol processor taps into Nectar's 100-Mbps fiber-optic media via an on-board network interface. All protocol-specific processing is performed using a general-purpose 16-Mhz SPARC CPU which runs a lightweight multitasking runtime system. In other respects, however, the CAB features a customized, yet protocol-independent, design. To avoid memory bandwidth bottlenecks, a specialized memory architecture is used. The architecture employs 35-ns static RAM divided into 512 Kbytes of program memory and 1 Mbyte of data memory. As with the VMP NAB, data memory and program memory are optimized for access by a DMA controller and a CPU, respectively. The custom DMA controller supports data transfer among CAB memory and the network interface and among CAB memory and host memory.

Application processes in the host communicate with the CAB using the Nectarine API. Nectarine provides low-latency access to the CAB through the use of a mapped shared-memory scheme that enables direct communication between the user and protocol-processor levels, bypassing the kernel level entirely. As part of the API initialization code, an application process at the user level maps CAB data memory into its address space using the `mmap()` UNIX system call. This system call, performed only once, invokes a kernel

device driver routine which facilitates the mapping. The application process may thereafter directly communicate with the CAB through Nectarine subprogram calls and the shared memory region. The primary advantage of this scheme is that it avoids the overhead of making system calls for each CAB request. It is not likely, however, that this approach can still provide security among application processes since they must share a common memory region. The library routines of the Nectarine interface do support synchronization for the shared region, so conforming processes may safely communicate with the protocol processor through the Nectarine API.

Due to the use of shared-memory, synchronization is necessary between host processes and the CAB. This is done via condition variables and *signal* and *wait* operations. Another form of synchronization is the indication of pending protocol processor status. To this end, an application process may receive notification of a condition via polling at the cost of busy waiting. Alternatively, the process may choose to sleep on the condition and receive an application interrupt when the condition changes; however, this approach requires kernel support and therefore incurs the cost of a system call.

The Nectarine API supports command, status, and data streams between the protocol processor and application process. The command stream to the protocol processor is the *CAB signal queue*, and the status stream to the host is the *host signal queue*. A host process can notify the CAB of pending commands via an interrupt. Similarly the CAB may interrupt processes sleeping on the arrival of status; otherwise, such applications poll as discussed above. The data stream between an application process and the CAB is via *mailboxes*, shared memory regions accessed through a synchronization protocol. Mailbox writers must delimit their writes to mailbox memory with `Begin_Put` and `End_Put` operations; similarly, readers must delimit their reads with `Begin_Get` and `End_Get` operations. Some undefined scheme maps commands and status to the appropriate mailboxes containing their associated data, if any.

Cooper et al. built a complete prototype and therefore provide actual performance measurements of its use. These were observed at both the protocol processor (CAB) and user (host) levels using several of their protocols. For a short message over RMP, CAB-to-CAB latency is 241 μs and host-to-host latency is 414 μs. For large messages over RMP, CAB-to-CAB throughput is around 90 Mbps and host-to-host throughput is 30 Mbps. For the same message size, TCP gets about 35 Mbps from CAB-to-CAB and 24 Mbps from host-to-host. TCP's lower throughput is largely due to its checksum, which, in contrast to the Protocol Engine and VMP NAB, receives no custom VLSI support with the CAB. RMP does not include checksums; rather, it is optimized for the Nectar LAN, relying on its data-link error detection. The large performance drop between the host-to-host and CAB-to-CAB observations is blamed on the overhead of transfers across the host's VMEbus.

3.5. Netravali et al.

Netravali et al. developed an communications architecture in which their "SNR" protocol runs off-host and in parallel on a set of loosely-integrated, dedicated processing nodes [35]. They use a protocol architecture with a null network layer. A novel aspect of their architecture is that, to minimize bus contention, their planned design connects protocol processing nodes via a hierarchy of buses, each of which may be mastered independently. A "transmitter" sub-bus contains transmitting processor boards and a shared memory board; similarly, a "receiver" sub-bus contains receiving processor boards and another shared memory board. The two sub-buses are connected to another bus which acts as a backbone and allows access to a Network Interface Board (NIB) and the host computer system. The transmitting and receiving processors are general-purpose, single-board computers, but the NIB requires SNR-specific custom circuitry.

The NIB is a separate circuit board which not only transmits and receives TPDU frames to and from the network, but also has the ability to distinguish those which are for data or control. In addition, it manages queues for free TPDU buffers, data and control

TPDUs enroute to the network, and received data and control TPDUs. The NIB performs TPDU checksums as well.

The command, status, and data streams from the host computer to the protocol processor nodes are via two separate command and status FIFOs for each simplex connection. The impression from the literature is that the host must perform segmentation and reassembly of TSDUs and TPDUs, respectively. In this scenario, the host sends TPDUs to the protocol processor by acquiring a free buffer for its connection (apparently resident in host memory) and copying its data to the buffer (presumably with programmed I/O). It then commands the processor to send the TPDU by placing a pointer to its buffer in the connection's own transmit FIFO. A separate host transmitter process drains each connection's FIFO by transmitting its TPDUs using the NIB. TPDU reception is handled similarly. The host reads protocol processor status via a FIFO providing pointers to sent TPDUs and received TPDUs for the connection. The received TPDUs are queued in the proper sequence of a TSDU. The host apparently polls the status queue, since there is no mention of the use of interrupts in the literature. A host receive process copies incoming TPDUs to host memory.

Protocol processing is divided into the work of transmitter and receiver processes, each of which may execute on its own processor board or as a scheduled task on a single processor. Each transmitter or receiver process manages only one connection. All connection state resides in the shared memory board on the transmit sub-bus, and other global state information is stored in the shared memory board on the receive sub-bus. One transmitter process manages TPDUs for the connection, another handles retransmissions, and another queues status for the host. One receiver process ensures proper ordering and error control, another frees completed TPDUs and advances the flow control window, and another periodically sends transport protocol control frames indicating receiver status to the transmitting end of the connection.

In contrast to the eventual plans for a hierarchy of buses, the prototype discussed in the literature places all nodes on a single VME backplane. Each protocol processor node as well as the host, is a general-purpose Single-Board Computer (SBC) containing a 20-Mhz Motorola MC68030 and 4 Mbytes of local RAM. Each SBC runs a simple real-time multitasking operating system. In the prototype, one SBC was used to simulate a NIB, and a shared memory board was used to hold the NIB FIFOs and TPDU buffer memory.

The best performance results were received when two SBCs were used for all receiving processes and one SBC was used for all transmitting processes. To factor out the contention overhead inherent in their single-bus prototype, all measurements were performed by transmitting only TPDU headers and no data. As a result, estimates of the per-TPDU processing throughput of the architecture are highly speculative. According to the literature, the protocol processor prototype can handle a rate of about 10,000-15,000 TPDUs per second. They expect a higher rate with a full hierarchical-bus implementation of their architecture.

3.6. Beach

Beach presents the design of UltraNet, a commercial architecture aimed at providing near-gigabit data transfer between supercomputers over gigabit links [3]. To achieve such high performance, UltraNet architects use an off-host firmware implementation of a modified TP4 and employ up to two general-purpose processors and a small amount of custom VLSI support. The network layer of this architecture is null. In addition to linking supercomputers, their architecture also connects workstations, albeit with lower-speed links (250 Mbps). To meet these varying demands, they provide two models of their protocol processor, one for supercomputers and another for UNIX workstations.

The designers chose TP4 as their transport protocol for its simplicity. Beach makes a point that the protocol processor’s high performance (discussed below) with TP4 shows that lightweight transport protocols are not necessary for high-speed protocol processing.

This is inconsistent with the fact that his architects made radical lightweight modifications to TP4, such as a simplified TPDU format, reduced options, placement of checksums in the TPDU trailer rather than the header, expanded maximum TPDU size (32 KB), and the addition of selective acknowledgments. Thus, such a conclusion is simply not warranted.

TP4 processing is handled by two general-purpose microprocessors on the supercomputer model of the protocol processor, the Control Processor (CP), an 8-Mhz AMD 29300 and the Data Acknowledgment and Command Block Processor (DACP), a 12-Mhz Intel 80386. For the workstation model, a single 16-Mhz 80386 acts as the CP and also performs the processing of a Virtual DACP (VDACP). The DACP performs common case processing such as the handling of data and acknowledgment TPDUs and the processing of commands from the host. Its software is highly optimized; for the supercomputer model, the DACP is programmed with hand-written microcode, and for the workstation model, the VDACP executes hand-coded assembly language. The CP handles complex and/or rare protocol processing tasks such as connection setup and error control, and is programmed in a high-level language. The CP and DACP communicate via the shared memory of a connection database. A very lightweight multitasking operating system is run on the two processors. TP4 checksums are performed “on-the-fly” in custom hardware as the TPDU flows through the on-board network interface. This is facilitated by the movement of the checksum field into the TPDU trailer.

The model of the protocol processor for UNIX workstations resides on the host’s VMEbus, over which inter-processor communication occurs. The command and status streams between the protocol processor and the host are constructed as two queues of command blocks resident in host memory. Notification of pending commands and status is performed via hardware interrupts. The architecture is carefully designed to require only one copy of each TSDU in the common case. This ensures higher throughput and eliminates the need for the large pool of high-speed buffer RAM that would be required to sustain near-gigabit transfer rates. The copy is performed using a DMA controller on the protocol pro-

cessor. Application processes running at the user level use a Berkeley socket API to perform communication. The API code then invokes protocol processor device driver routines at the kernel level through system calls.

Throughput results are presented in the literature for both supercomputers and workstations. When sending very large messages (on the order of a megabyte) between a Cray X/MP and a Cray 2, presumably over a 1 Gbps link, a throughput of 384 Mbps is achieved. Between two Sun-3 workstations, their results show a peak rate of 31 Mbps. Counter to intuition, the peak Sun-3 performance is higher than that between Sun-4 workstations, even though the Sun-4 machines have faster CPUs and VMEbus transfer rates.

3.7. MacLean and Barvick

MacLean and Barvick developed an off-host Protocol Accelerator (PA) board whose goal is to eliminate the bottlenecks in transport protocol processing. The PA approach is to use two general-purpose microprocessors and additional custom hardware to execute a modified TCP [31]. Their communications architecture has a null network layer and serves a UNIX host computer containing a 25-Mhz Motorola MC68030 CPU.

A VME backplane contains the PA board along with the host processor board and a 4-Mbyte host memory board. The protocol processor contains two 25-Mhz Motorola MC68020 microprocessors; one is for receive processing, and the other is for transmit processing. To avoid microprocessor bottlenecks, no operating system is used, and each processor has its own supporting 128 KB of RAM, I/O ports, and interrupt circuitry. An additional 128 KB pool of 40-ns static RAM is used to contain TCP Transmission Control Blocks and is shared, facilitating communication between the two processors and the host.

The data stream is carefully designed to avoid bottlenecks. The network interface resides on the PA board, and there is a direct data path between host memory and the network interface which prevents intermediate buffering of transmitted application data units. In addition, transfers between host memory and the protocol processor are performed by

one of two DMA controllers, one for sending and one for receiving. These controllers support scatter-gather DMA and, according to the literature, provide a transfer rate of up to 264 Mbps. Two hardware checksum components operating with the DMA are provided for sending and receiving. To use the components efficiently, the TCP checksum field was moved to form a TPDU trailer. For unspecified reasons, certain IP header information was also added to the TCP header. MacLean and Barvick claim that all the hardware components can operate in parallel.

Application processes use the PA through a BSD socket API which, in turn, makes system calls to routines in two UNIX kernel device drivers, one for the transmit processor and one for the receive processor. The command and status streams between the protocol processor and host are presumably implemented via the fast, shared memory, and hardware interrupts are used to inform the host of command completion.

The host-to-host throughput for 20-KB TPDUs is quoted in the literature as 11,000 TPDUs per second, and the PA-to-PA performance is 15,000 TPDUs per second. These results are with 252-byte frames over a loopback network interface which has a 320 Mbps transfer rate. Since the interface is loopback, a single host must both send and receive TSDUs. To minimize the effect of this double duty on bus contention, received data is not copied over the bus in their tests. The architects blame the host-to-host performance loss on UNIX overhead, which increases with TSDU size. They claim that this overhead is due to the scheduling of other host processes.

3.8. Mitchell et al.

Mitchell et al. discuss their plans to develop an off-host communications architecture for embedded systems by executing the transfer layer protocol, XTP, on an attached processor board [38][33][34]. Their design goal is to ensure high throughput for both the protocol and the application tasks on the host. The host system has a Motorola MC68030 microprocessor, runs a real-time operating system, and resides, along with the protocol pro-

cessor, in a VME-like proprietary backplane bus. At the time the documents on the architecture were written, not all design issues were fully resolved. The discussion below briefly reviews the tentative design described in the literature. No host-to-host performance estimates were available.

The protocol processor board is designed to be made up of a custom integration of stock microprocessor, memory, and network interface components. An Intel 80960CA was chosen as the microprocessor. It executes a commercial version of XTP, modified to execute off-host and on the 80960. To maximize the performance of data transfer, the designers hoped to utilize some form of high-speed static RAM. An on-board network interface chipset implements an FDDI-like proprietary MAC protocol.

Plans for the command and status streams were tentative. Application processes command the processor through a socket-like API. The command and status streams are implemented as queues in shared memory containing control blocks. To signal pending commands, the host interrupts the protocol processor, and to signal pending status, the protocol processor interrupts the host. Data are transferred between the host and protocol processor using a DMA controller built into the 80960 microprocessor.

3.9. Siegel et al.

Siegel et al. develop an architecture which seeks to overcome the bottlenecks in transport protocol processing [39]. They approach the problem at two ends by employing (1) a simplified version of TP4 and (2) an off-host architecture in which TP4 runs in parallel on a set of general-purpose processors aided by custom VLSI.

First, we briefly discuss the changes to TP4 because they facilitate the use of the custom VLSI support. The modifications eliminate segmenting TPDUs and multiplexing TP4 connections over a single network connection. In addition, the flow control is simplified and the extended PDU numbering format is mandated.

The protocol processor consists of a tight integration of several general-purpose RISC microprocessors and custom VLSI circuits. There are separate processors for the sending and receiving data streams, each with its own local memory. In addition, a custom VLSI processor coupling device allows synchronized processor access to shared memory, promoting inter-processor communication. Other custom circuits support timer and buffer management and perform checksums. An “on-the-fly processor” detects the end of TPDU headers streamed in from the network interface in order to route the header and data to separate buffers. These buffers are separated in order to optimize the use of the memory. Headers are stored in high-speed 30-ns static RAM since they will be critically manipulated by the protocol processor. Data is stored in high-capacity 70-ns DRAM since it arrives in large quantity and need only be accessed in a single copy.

With respect to host and protocol processor communication, only the data stream and processor synchronization mechanisms are described in the literature. For the data stream, copies are performed with a special DMA controller that automatically allocates a destination buffer from a pool as part of the transfer. In addition, the architecture is flexible with respect to the notification of the host of pending status. Interrupts are recommended for hosts with fast interrupt handler context swap times. If interrupts are inefficient on the host, an alternate scheme involving a pollable, shared status memory can be chosen.

3.10. Summary

We have surveyed several efforts to develop off-host communications architectures. The designs share several common characteristics:

- Use of lightweight protocols, or common protocols (e.g., TCP) modified to be more efficient
- Attempts to harness the inherent parallelism in protocol processing
- Optimized common-case protocol processing
- High-speed data movement hardware
- Minimized buffering and data copies

- Optimized memory architecture for fast access to protocol data

The protocol processors tend to differ in the degree to which they use custom components. Chesson's Protocol Engine represents a highly-custom and tightly-integrated approach using several custom VLSI circuits. At the other extreme, Mitchell et al. integrate stock hardware of a general nature (e.g., processor, memory, DMA controller, and network interface). In the next chapter, we will describe our architecture, which is even more general-purpose and loosely integrated. It combines commercially-available hardware boards to produce a custom system and, for the most part, executes previously-available software.

4 An Off-Host Communications Architecture for SAFENET

4.1. Overview

This chapter presents the design of an off-host communications architecture that executes a transfer layer protocol, XTP, on a single-board computer attached to a host computer system. As a whole, the communications architecture is designed to implement the SAFENET lightweight protocol suite. The design of this system illustrates our efforts to resolve the design issues presented in chapter two. It is this system that will be studied throughout the remaining chapters of this thesis.

4.2. SAFENET

The United States Department of Defense has adopted SAFENET (Survivable Adaptable Fiber Optic Embedded Network) as its protocol architecture for mission-critical computer systems [21][36][32]. SAFENET is unique in that it specifies a dual-protocol stack. A suite of ISO protocols is specified for use in non-real-time systems (e.g., file transfer and electronic mail), and a so-called Lightweight Profile defines the protocols to be used for latency-sensitive applications. With the ISO suite, the goal is to maximize interoperability, whereas with the "lightweight" suite, the goal is to optimize performance. In the lightweight stack, the presentation and session layers are null, and transport services are provided by XTP running over LLC, SNAP, and FDDI-based MAC and physical layers. The application layer provides very general communication services that map closely to XTP functionality. Figure 4.1 illustrates the SAFENET architecture.

It is important to note that due to the design of the Lightweight Application Services (particularly its null presentation and session layers), an ASDU has the same format as a TSDU. As a result, we often shift terminology when describing the flow of such data

through our architecture. The reader should not be concerned with this, as the distinction between such data units is not particularly important in our case.

SAFENET is a part of the Next Generation Computing Resources (NGCR) program, an effort which attempts to break from the former practice of developing highly custom government systems at great cost. Rather, the NGCR idea is to set standards specifying minimal requirements and then employ commercially-availably technology meeting such standards whenever possible. This new way of developing government systems cuts design time and allows wide market support, thereby promoting higher quality and lower cost. The NGCR philosophy had an impact on the design of our architecture as is discussed below.

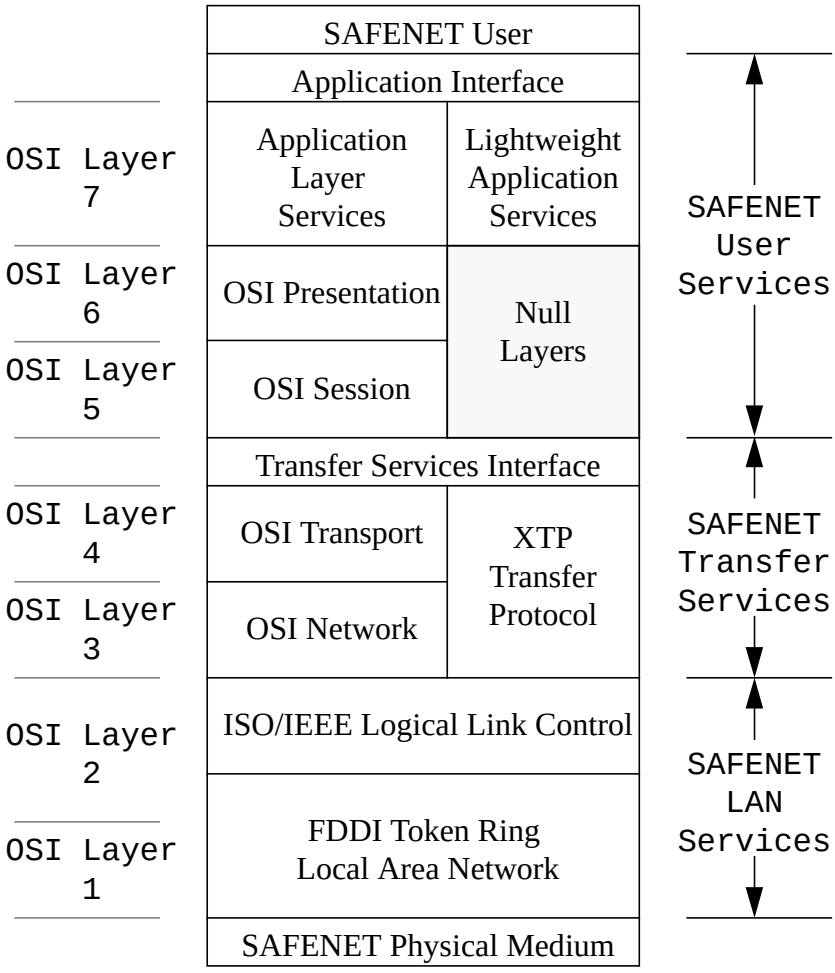


Figure 4.1: SAFENET Protocol Architecture

The Computer Networks Laboratory at the University of Virginia was awarded a Navy contract to develop an implementation of the SAFENET lightweight protocol suite for the Desktop Tactical Computer (DTC-2) host, including XTP, an FDDI MAC interface, an Ada API, and directory services. The resulting hardware architecture, Ada API, and XTP implementation are described in detail in [13][14][15]. The technical design decisions that resulted in our architecture are discussed below.

The fundamental design choice that drove the rest our implementation efforts was to use an off-host communications architecture. This decision was motivated by the characteristics of the applications served by the lightweight stack. The mission-critical applications using the Lightweight Application Services have real-time requirements, i.e., they need minimal execution time penalty from the service primitives as well as high throughput and low latency from the protocol. With the aim of providing such performance-critical service, we chose to use an off-host architecture. This approach offered the opportunity to gain some of the potential benefits of off-host protocol processing outlined in chapter one; however, the actual effects of off-host implementation were not at all clear from the outset. Thus, the determination of what benefits our off-host architecture would actually deliver was a fundamental research question.

4.3. Host Computer System

The host computer system served by our off-host communications architecture is the C3 Computer Systems Desktop Tactical Computer 2 (DTC-2). The machine is based upon a Sun 4300 motherboard including a 25-Mhz SPARC CPU and 32 Mbytes of DRAM. The motherboard resides in the first slot of a VME backplane bus into which additional hardware device modules may be inserted. Our DTC-2 runs the SunOS 4.1.1 operating system, a variant of BSD UNIX.

4.4. Architectural Constraints

The host hardware and operating system placed several constraints upon the design of our communications architecture. The layout of the host system's motherboard was fixed in that no modifications to it were possible. This forced a more loosely-coupled design for our communications architecture than could have been the case if we had the opportunity to architect the motherboard; with the DTC-2, any additional off-host communications hardware would have to be connected to the host system via its I/O backplane.

The bus and motherboard architecture also forced certain other design constraints. Since the host motherboard resides in the first slot of the VME backplane, it is required by the VMEbus specification to act as the bus arbiter [2]. This constrained our bus arbitration policy since the Sun 4300 motherboard supports only a fixed-priority arbitration policy [45]. In addition, the board supports only a Release On Request (ROR) bus release strategy. A very significant restriction imposed by the motherboard is that only a restricted region of its virtual address space is addressable on the VMEbus. The effect of this restriction is discussed in section 4.5.3.2.

The SunOS 4.1 operating system's use of a privileged kernel with a protected address space acted as an obstacle in our architecture. SunOS provides security between application processes running at the user level, the operating system running at the kernel level, and hardware devices at the backplane bus level through the use of disjoint virtual address spaces. To ensure security, user processes do not normally have the ability to address the memory of other processes and the kernel. Likewise, they are also barred from direct access to shared hardware resources such as I/O device registers; rather, they must invoke the kernel to access such resources. Reference to I/O devices on the SPARC processor is possible only in the following manner. All SPARC access to I/O devices must be via memory-mapped I/O; therefore, to communicate with a device, its registers or memory must be properly mapped into a virtual address space. With SunOS, only the kernel has the ability to create virtual address mappings, so it alone may enable the use of hardware

devices. These constraints implied that our architecture would be forced to have a kernel level, and that such a level would play a key role in our architecture.

4.5. Design Choices

Within the bounds of the constraints discussed above, we were otherwise free to design our off-host communications architecture as we saw fit. Our choices are initially discussed in terms of hardware and software layers. Since the hardware choices dictate the software choices, we discuss the hardware first. The most important design issues are those of the inter-layer communication mechanisms; these are discussed last.

4.5.1. Hardware Components

As mentioned above, any additional hardware in our system would have to reside in modules on the DTC-2's VMEbus. This still left us with many options. We could have attempted an architecture that was as tightly integrated as possible while still using an I/O bus. With such an architecture we could have used a single additional module containing custom VLSI to implement XTP functionality and an on-board network interface. Such a highly-integrated approach, however, would require great development cost in terms of both time and money. Unfortunately, such development-intensive design was inconsistent with the philosophy of the NGCR program from which SAFENET was spawned. In accord with the NGCR philosophy we were forced to seek a more economical solution. Our aim was to integrate a set of commercially-available hardware components which could meet our needs. A single module containing a more loosely-coupled organization of hardware was considered. This hardware contained such components as a high-performance, yet general-purpose, microprocessor and an on-board network interface. However, at the time, the development cost for this approach was deemed too high.

We finally opted for a very loosely-integrated hardware architecture in which we would employ two additional VMEbus modules, one containing a single-board computer (SBC) that would act as the protocol processor, and another containing an FDDI network

interface. For the SBC we chose a Motorola MVME-167A (167). Although the 167 is a general-purpose SBC, it still comes moderately well suited for protocol processing; the board features a 25-Mhz Motorola MC68040 microprocessor, 8 MB of on-board 70-ns DRAM, and a DMA controller. The on-board memory is addressable in the local address space of the SBC and may also be mapped into VMEbus address space. The 167 is capable of mastering the VMEbus and may employ either a Release On Request (ROR) or Release When Done (RWD) bus release policy. For the 167 we chose the pSOS+ lightweight multitasking operating system. With pSOS+, multiple tasks may run on the processor, all sharing a single address space.

The network interface supports the MAC layer and below of FDDI and is manufactured by Network Peripherals (NP). This board is controlled by a device driver using I/O registers. It also features a DMA controller, which allows it to master the VMEbus and handle data transfers without incurring device driver CPU cycles. The controller is capable of using the block transfer mode of the VMEbus to achieve higher throughput. The network interface and its device driver exchange data frames through a pair of separate send and receive hardware FIFO registers.

The necessary low-level mechanisms exist to allow communication between each board on the VMEbus. The host, protocol processor, and network interface are all capable of mastering the bus and addressing memory or registers in VMEbus address space. Furthermore, each board may act as a slave in order to respond to read or write bus transactions. In addition, each board may issue VMEbus interrupt requests. Only the host and protocol processor, however, may handle such interrupts. Figure 4.2 depicts our communications hardware architecture.

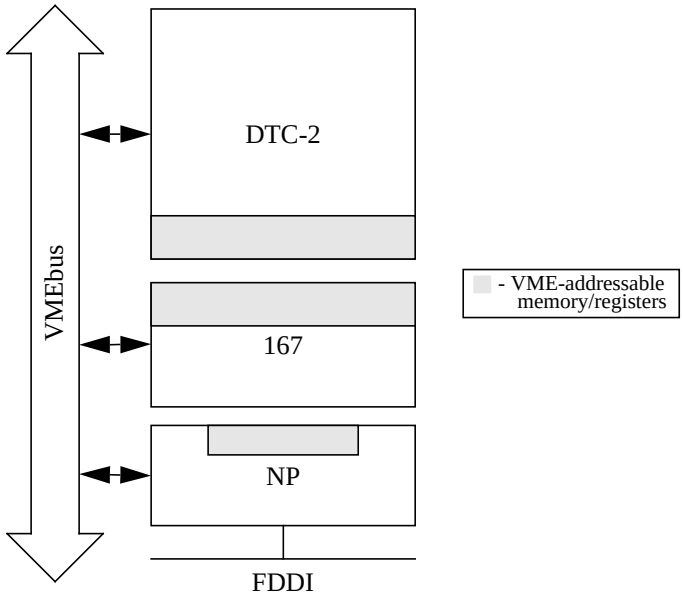


Figure 4.2: Communications Hardware Architecture

4.5.2. Software Layers

As a result of our hardware and operating system environment, our architecture consists of several levels. Those at which software may reside are the user and kernel levels of the host, and the protocol processor level. Each level has its own address space, which serves to complicate our design. Our remaining design choices deal with how to distribute the software implementing SAFENET functionality at each level and achieve the inter-level communication for our command, status, data, and options streams. This section discusses the distribution of our software, and the following section discusses the communication streams. Figure 4.3 illustrates our overall system architecture.

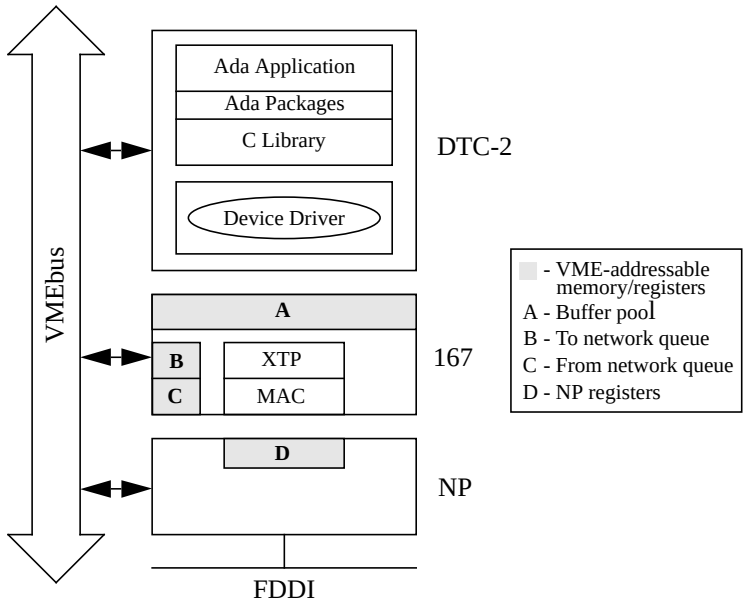


Figure 4.3: Communications System Architecture

Our user-level software design was largely dictated by Ada and our operating system. As required by the API specification for the SAFENET Lightweight Application Services, our software architecture must provide communications services to Ada application programs. Given our Ada compiler, these applications must run at the user level as SunOS processes. Furthermore, the Lightweight Applications Services API was given as a set of Ada package specifications. Therefore, at least part of our software was constrained to be written in Ada. However, due to the fact that we were developing systems software for UNIX (a system with a large bias toward the C programming language) we felt that the majority of the API implementation would be most practically written in C. We therefore chose a software architecture in which the API is implemented with a thin layer of Ada package bodies that employ lower-level software in a library of C code linked with each application.

At the kernel level, we developed a UNIX character device driver. As mentioned in section 4.4, we were forced to have a kernel level in order to handle at least some hardware-

intensive functions. Specifically, a kernel-level driver has two unique characteristics which allow it to effectively operate the protocol processor. First, it has access to the system's paging hardware and therefore has the ability to map virtual memory regions. This allows it to access both the user-level address space and the VMEbus address space. Second, it uses the privileged supervisor mode of the SPARC CPU. As a result, it may carry out privileged hardware-oriented instructions. For example, only the kernel can set the interrupt priority level of the processor, and only kernel-level ISRs may handle hardware interrupts.

Beyond the use of the kernel for these hardware-intensive operations, we made the choice to have the kernel manage all protocol processor operations. That is, although it was possible through the use of user-level memory mapping to allow user processes to control the protocol processor directly, we chose to only place this responsibility on the kernel. The drawback to this is that it requires user processes to issue system calls for each protocol processor request. The advantage is that the device driver may act as an honest broker, providing secure access to the protocol processor. In addition to providing security, it also provides synchronization in that, since it alone interacts with the protocol processor, all user accesses to protocol processor are serialized.

As a result of these design choices, the kernel level device driver is responsible for handling every operation on the protocol processor device. Such operations include writing to and reading from it its memory or registers and handling its interrupts.

The protocol processor level contains the remainder of our software. This consists of a commercially-available, multi-threaded implementation of XTP and a MAC level device driver for the FDDI network interface. The XTP protocol and MAC driver run as a set of pSOS+ tasks sharing the single address space of the 167. There were no real software choices with respect to XTP since no other commercially-available implementation of XTP was suitable for the pSOS+ environment. The decision to have the MAC driver run as a separate task was dictated by the XTP implementation.

4.5.3. Inter-Layer Communications Mechanisms

Given our hardware and choices of software placement, the issue of how to pass command, status, synchronization, and data information between the user, kernel, and protocol processor levels was left open. Our approach to this problem revolves around the design of the command and status streams. We chose to encapsulate command and status information into two separate data structures. A *control block* indicates a user process' command, and an *acknowledgment block* (so named because it usually acknowledges command completion) indicates the protocol processor's status. Both of these data structures contain pointers to associated TSDU data, if any.

The API features several command options which are used in protocol initialization or connection setup calls; however, such information is not required in the common case of data transfer. Rather than pass this information in each control and acknowledgment block, we chose to include it in a separate options stream. With this scheme, control blocks and acknowledgment blocks contain pointers to their options, if any; thus, such blocks are kept smaller in size, decreasing their overhead in copies.

4.5.3.1. User-Kernel Communication

As mentioned above, in order to ensure secure access to the protocol processor, we allow only the kernel to manipulate the protocol processor device. Thus, only the device driver may issue control blocks to the device and read acknowledgment blocks from it. As a result of this decision, the C library code of a user-level application must invoke the kernel-level device driver in order to perform its communications operations. This user-kernel communication is performed through use of a combination of system calls and shared user- and kernel-level memory. In order to issue a command, the C library code constructs a control block indicating such information as the command code, the connection on which it is to be performed, and any associated options or data. The user-level library code then performs a `write()` system call containing a pointer to the control block as a value parame-

ter. Similarly, to read status the process issues a `read()` system call specifying a pointer to an acknowledgment block. When the `read()` completes, the C library code may examine the contents of the block. Only the pointers to such blocks are actually copied to kernel address space; the information in the blocks is simply mapped into kernel address space. The data and options streams cross the user- and kernel-level boundary through a similar shared memory mapping scheme.

Another issue is the notification of an application process in order to indicate pending status from the kernel level. For this, we chose to use a combination of both polling and user-level interrupts (UNIX signals). The reason for such complexity is that the Ada API supports two modes of primitive calls, *synchronous* and *asynchronous*. A synchronous API primitive call issues a command and blocks on the arrival of status before returning to its caller. On the other hand, an asynchronous call simply issues a command and does not wait on its status. The asynchronous call does, however, return a handle on its execution status known as an *activity index*. A user-level interrupt is used to indicate the completion of a command to the API implementation. Its arrival causes the new execution status of the primitive call to be indicated in an internal API data structure referenced by the activity index. To discover when an asynchronous call has completed, an application must poll the state of the call using its index. For a synchronous call, the situation is similar, except that the polling is performed automatically within the call so no activity index is necessary.

More efficient and elaborate methods of implementing synchronous and asynchronous primitive calls were considered. For example, rather than forcing an Ada task to busy wait on the completion of a synchronous command, we considered blocking the task until an interrupt for the command arrived; however, our Ada compiler did not support low-level scheduling operations on tasks, prohibiting this approach. For the notification of the completion of an asynchronous call, we considered tying the arrival of a user-level interrupt to the execution of a application-specified interrupt handler. This scheme was prohibited by Ada limitations on using subprograms and task entries as first-class programming language

type entities and by the inherent problems of calling Ada subprograms from a C-based interrupt stack.

4.5.3.2. Kernel-Protocol Processor Communication

We chose to perform communication between the kernel and protocol processor levels through shared memory and interrupts. At system boot time, the kernel device driver maps in a large region of the local memory on the protocol processor board into its virtual address space. This shared memory region is used to contain two data structures, the To Network (TN) and From Network (FN) queues. These act as the command and status streams. The host CPU and protocol processor access these queues following a discipline which promotes synchronized access to their shared memory. The kernel level device driver writes control blocks to the TN queue and reads acknowledgment blocks from the FN queue. In contrast, the protocol processor reads control blocks from the TN queue and writes acknowledgment blocks to the FN queue. In addition to the use of the two queues, the protocol processor uses interrupts to indicate to the host that is has a pending acknowledgment block.

Incoming status at the kernel level causes a user-level interrupt (a `SIGIO` signal) as discussed above. The following demultiplexing mechanism ensures this. The kernel-level hardware ISR that runs as a result of a protocol processor interrupt must issue a signal to the application for which the acknowledgment is bound. In order to route the signal to the proper destination process, the device driver maintains connection state information which relates the connection identifier in a field of the incoming acknowledgment block to its owning process. This demultiplexing operation incurs some degree of overhead, as does the maintenance of connection status.

In order to avoid bottlenecks, the design of the data stream (more so than the other streams) should avoid copies. With this in mind, we wished to avoid buffering TSDUs on the protocol processor. In order to accomplish this, we would need to provide the protocol

processor with direct access to host memory. Since the protocol processor may only access external memory that is addressable on the VMEbus, we would have to make host TSDU memory regions addressable in VMEbus address space to avoid buffering them. Unfortunately, the design of the Sun 4300 motherboard places unwieldy restrictions on what region of its memory is addressable in VMEbus address space. There is only one such region; it is called Direct Virtual Memory Access (DVMA) space, and it functions as follows. The motherboard maps the highest megabyte of kernel virtual address space into the lowest megabyte of VMEbus address space. This shared region of memory acts as a window for VMEbus masters to access the virtual memory of the host. In addition to being addressable by masters on the VMEbus, DVMA space may be accessed by devices on the motherboard that are capable of DMA transfers, e.g., the on-board Ethernet interface. Due to its unique status, DVMA space is a precious resource that must be carefully shared with other devices. In addition, user-level memory must be carefully mapped or even copied in order to place it in DVMA space.

Rather than deal with these restrictions, we opted to abandon our attempts to allow the protocol processor to address host memory. This decision had a dual impact on our design. First, it forced us to buffer all TSDUs in the protocol processor's on-board memory. Second, we could not use the DMA controller on the protocol processor board to copy the TSDUs because, in order for the controller to transfer such data structures, it must be able to address them. As a result, we were forced to use the host CPU for programmed I/O transfers of TSDUs between host memory and buffers on the protocol processor board. These same restrictions also implied that the control blocks, acknowledgment blocks, and options must be transferred between the host and protocol processor using such a programmed I/O mechanism.

The choice of which processor should allocate buffers was another design decision. Data buffers for the data stream are allocated by the kernel-level device driver in the host from a 4 MB buffer pool stored on the protocol processor. We found it much easier to have

the driver allocate this memory since synchronization would be required in order to have the allocation performed by the protocol processor. TSDUs enroute to the network are copied to these buffers from the user-level memory on the host as discussed above. Similarly, TSDUs enroute to user-level memory on the host are also copied from the protocol processor's buffer pool. Buffers for protocol options are allocated in the same fashion as data buffers.

The implementation of XTP places a limit on how many commands may be concurrently outstanding on a single connection. In our implementation, up to four sending operations may be performed on a connection, along with up to four receiving operations. In order to avoid exceeding these limits, the API implementation keeps a count for each connection and blocks the progress of any API primitive that attempts to exceed this limit.

4.5.3.3. Protocol Processor-Network Interface Communication

As mentioned above, the protocol processor contains an FDDI MAC device driver. This driver issues commands to the network interface via VMEbus write transactions to its control registers. Likewise, the protocol processor reads network interface status thorough the use of read transactions with the interface's status registers. TPDU frames are sent from the local buffer memory on the protocol processor to a transmit FIFO register on the network interface. The transfer of a frame is performed using the block transfer mode of the VMEbus by the DMA controller on the network interface. The DMA controller interrupts the protocol processor to inform it of transfer completion. The arrival of a frame is indicated to the protocol processor via an interrupt. The transfer of the frame from a receive FIFO on the interface to a buffer on the protocol processor is also done with block mode DMA, and an interrupt indicates DMA completion here as well.

4.5.4. A Data Flow Example

In order to illustrate how our design operates, we trace the data flow due to a single synchronous API call through our architecture. To start the data flowing, an Ada application

makes a call to an API primitive such as `SEND_MESSAGE` from one of our Ada packages in order to perform a transmitting operation. The Ada `SEND_MESSAGE` procedure invokes a routine called `c_send_message()` in the library of C code, which begins the C implementation of the API primitive. Aside from the management of some user-level resources and state information, the function of this C code is to build a control block for the command including the connection with which it is associated and a pointer to the ASDU referred to in the `SEND_MESSAGE` call. The library code then issues the control block to the kernel-level device driver with a `write()` system call.

The device driver begins execution at its `write()` entry point. It examines the control block to verify the user's right to issue commands on the connection. Then the driver acts upon the command, maintaining a small amount of state information. The driver allocates a buffer to hold the TSDU and copies it, via programmed I/O, to the buffer pool on the protocol processor using the kernel's `copyin()` utility function. It then enqueues the control block on the TN queue with `copyin()` and returns control to the calling process. Since the API call is synchronous, the user process busy waits on the arrival of an acknowledgment block for its command. Meanwhile, the protocol processor dequeues the command and performs the appropriate protocol operations to send its associated TSDU. The protocol processor transmits the TPDUs of the packet over the VMEbus and through the FDDI network interface using the network interface's DMA capabilities.

Once it discovers (via an acknowledgment) that the TSDU arrived at its destination, the protocol processor queues an acknowledgment block for the aforementioned `SEND_MESSAGE` control block on the FN queue and interrupts the host. This interrupt invokes a device driver interrupt routine which in turn signals with `SIGIO` the user process owning the connection identified in the acknowledgment block. A signal handling routine for `SIGIO` in the C library code of the user process performs a `read()` in order to dequeue the acknowledgment on the FN queue. The kernel then transfers the acknowledgment block into user-level memory with the `copyout()` kernel utility function. The signal han-

dling function uses the acknowledgment block information to cease the busy wait of the process. The signal handler also performs another `read()` to check for additional acknowledgment blocks and, finding none, returns from the interrupt. When the user process resumes execution, it discovers the completed status of the `SEND_MESSAGE` call, thereby finishing its execution.

5 Performance Analysis

5.1. Overview

In this chapter we study the overall performance of our off-host communications architecture, profile it in detail, and identify the bottlenecks in its design. In order to illustrate the effects of our design's multilevel architecture, we provide throughput and latency measurements at several of its levels. To understand the reasons for our user-level performance, we include a profile of host processing time for the `SEND_MESSAGE` and `GET_MESSAGE` operations. We also include an assessment of the host processing required to perform such operations. From the performance measurements, profiles, and some other experimental results we identify the bottlenecks present in our architecture.

5.2. Performance Observed at Various Service Levels

Our architecture is built upon several layered levels of service, each of which has an effect on the performance of the service level above it. Note that the service levels we speak of here are not the same as the architectural levels discussed in chapter four. These service levels are aimed at describing the performance of the fundamental communications services accessible at the user and protocol-processor architectural levels. The following sections describe the communication performance of our architecture observed at three service levels, the FDDI MAC device driver level, transport protocol level, and the user level, for the full range of message sizes available at each. Using our terminology, a MAC-level message is a frame, a transport-level message is a TSDU, and a user-level message is an ASDU[1]. All results were obtained using time-average measurement techniques, i.e., the time for several iterations of an operation was measured, and the resulting average time for a single such operation is reported.

1. The user level would perhaps be more aptly termed the application level.

5.2.1. MAC Level

The MAC level provides the transport protocol with a raw, frame-oriented, data link service over the 100-Mbps FDDI network. These results were obtained using a pair of Network Peripherals FDDI boards driven by a pair of Motorola MVME 167A processor boards in two stand-alone VME card cages. During these tests, no operating system ran on the processors, and VMEbus transfers between the NP board and 167 were performed with block mode DMA. Figure 5.1 shows end-to-end latency, and Figure 5.2 shows throughput. Here, latency is half of the round-trip time of a frame, and throughput measures the rate at which the MAC driver can transmit frames with no receiver. A minimum latency of 91.5 ns occurs for a frame with no payload, and the maximum throughput of 56.6 Mbps occurs for frames carrying a payload of 4487 bytes (4500-byte maximum FDDI frame less LLC and SNAP headers).

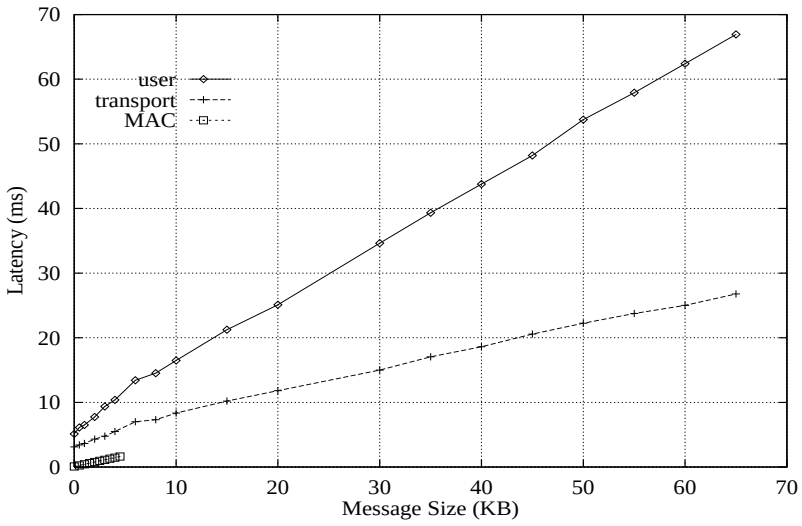


Figure 5.1: End-to-End Latency vs. Message Size

5.2.2. Transport Level

In our architecture, the transport protocol (XTP) provides reliable end-to-end delivery of memory buffers from the local memory of one protocol processor board to another.

The performance of this service was measured at the top-level interface to XTP, which is the interface used by our protocol processor device driver. The host system was in no way involved with these measurements; rather, they were performed by the protocol processor itself. For the throughput measurements, the data transfer operations were pipelined on a single connection using the streaming features of our protocol implementation, i.e., several TSDUs could be submitted to a connection at once. For all measurements, checksums were disabled with the XTP `NOCHECK` option, but rate control was unused. End-to-end latency and throughput are shown in Figures 5.1 and 5.2, respectively. Here, end-to-end latency measures half the round-trip time of an entire TSDU, and throughput is for reliable transmission. The minimum latency of the protocol is 2.7 ms for a one-byte message, and a maximum throughput of 23.8 Mbps occurs for a 64-KB message.

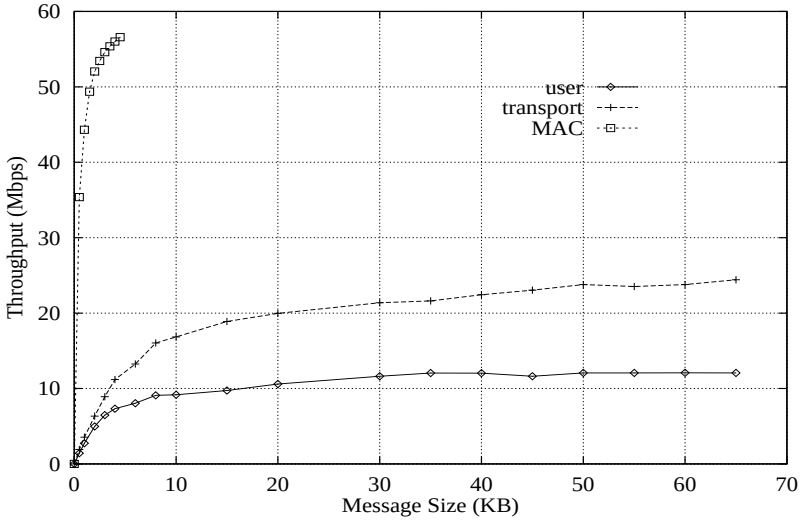


Figure 5.2: Throughput vs. Message Size

5.2.3. User Level

All user-level performance measurements were obtained using two Ada programs running on separate DTC-2 hosts using the connection-oriented `SEND_MESSAGE` and `GET_MESSAGE` primitives. Figures 5.1 and 5.2 show end-to-end latency and throughput

for the full range of ASDU sizes. End-to-end latency measures half the round-trip time of an entire ASDU sent between two Ada applications. For the throughput measurements, the communication primitives were performed asynchronously, and XTP's rate control features and NOCHECK option [44] were used to provide maximum performance. RATE was set to 1.5 MB/s and BURST was set to 10 KB/burst. For the latency measurements, NOCHECK was set, but rate control was unused. The minimum latency occurred at a message size of one byte and was 5.1 ms. The maximum throughput was 12.1 Mbps at a message size of 64 KB.

5.3. Performance Profile

To study why we observe the user-level results above, we profile the host CPU's execution of the SEND_MESSAGE and GET_MESSAGE primitives used in our user-level performance tests. In the next section, the analysis of the of this profile information will help to expose the bottlenecks in our design. The profile data was compiled using two methods. The user-level components of the profiles were measured with the UNIX prof utility. Since we had no tool which supported the simultaneous profiling of both the C and Ada code, we profiled only the C code. This choice essentially omits from our profile only a few Ada procedure calls, having no appreciable impact on our performance. For simplicity, all runs use the synchronous API calls. The performance results are derived from statistical measurements resulting from several thousand iterations of calls to the SEND_MESSAGE and GET_MESSAGE primitives. The kernel measurements were obtained differently. Since we had no profiling tools available for the kernel, measurements for its components were gathered using separate test programs employing the time-average measurement techniques discussed in Appendix A.

First, we concern ourselves with the processing of short ASDUs. Table 5.1 lists where the host processor spends its time during a SEND_MESSAGE call for a one-byte ASDU. The "other" category accounts for accumulated measurement error and operations

which consume less than 10 µs. The profile for a one-byte GET_MESSAGE operation is listed in Table 5.2; it is similar to that of a SEND_MESSAGE.

Operation	µs/call	Calls	total µs
wait for completion	3,881	1	3,881
`physio()` & `iodone()`	301	3	903
`read()` system call	81	2	162
signal delivery	143	1	143
`write()` system call	83	1	83
disable signals	25	3	75
enable signals	22	3	66
control block `copyin()`	29	1	29
ack block `copyout()`	23	1	23
other			732
total			6,097

Table 5.1: Profile of a One-Byte SEND_MESSAGE

The wait for completion is the amount of time that the host awaits an indication that the operation is complete. Although it accounts for most of the total time, the wait does not require host processor cycles; rather, it is a function of the transport protocol's performance. All other times in the table are incurred by various UNIX services; this overhead will be discussed in the next section. For now, it suffices to state that UNIX overhead dominates the host processing time when the ASDU size is small.

Operation	µs/call	Calls	total µs
wait for completion	4,045	1	4,045
`physio()` & `iodone()`	301	3	903
`read()` system call	81	2	162
signal delivery	143	1	143
`write()` system call	83	1	83
disable signals	25	3	75
enable signals	22	3	66
control block `copyin()`	29	1	29
ack block `copyout()`	23	1	23
other			568
total			6,097

Table 5.2: Profile of a One-Byte GET_MESSAGE

Note that the totals for sending and receiving are the same due to the interdependency of the SEND_MESSAGE and GET_MESSAGE operations and the fact that their exe-

cution is repeated in the tests. (Such repetition is necessary for our profiling technique.) These effects do not alter the quantities of interest to us.

To study the processing overhead of long ASDUs, we profile a `SEND_MESSAGE` of 64 KB in Table 5.3. As before, the total time is dominated by the wait for completion. However, with the long ASDU size, length-dependent operations such as the cost of allocating and deallocating a buffer on the local memory of the protocol processor become notable. Furthermore, a startling result is the time required to perform the `copyin()` of a TSDU across the VMEbus. Such overhead is discussed further in the next section. It is clear that, for long ASDUs, the time to perform the bus transfer dwarfs all UNIX overhead.

Operation	µs/call	Calls	total µs
wait for completion	23,455	1	23,455
data `copyin()`	18,164	1	18,164
`physio()` & `iodone()`	301	3	903
`read()` system call	81	2	162
signal delivery	143	1	143
get and return data buffer	109	1	109
`write()` system call	83	1	83
disable signals	25	3	75
enable signals	22	3	66
control block `copyin()`	29	1	29
ack block `copyout()`	23	1	23
other			1,636
total			44,848

Table 5.3: Profile of a 64-KB `SEND_MESSAGE`

The profile for a 64-KB `GET_MESSAGE` is listed in Table 5.4. It is similar to that of the `SEND_MESSAGE` except that the time for the data `copyin()` from host memory to protocol-processor memory is replaced with that for a data `copyout()` from protocol-processor memory to host memory. Note here the counterintuitive result that the host spends more time copying the received TSDU over the VMEbus than it does waiting for that processor to receive the TSDU from the network.

Operation	µs/call	Calls	total µs
data `copyout()`	23,712	1	23,712
wait for completion	17,525	1	17,525
`physio()` & `iodone()`	301	3	903
`read()` system call	81	2	162
signal delivery	143	1	143
get and return data buffer	109	1	109
`write()` system call	83	1	83
disable signals	25	3	75
enable signals	22	3	66
control block `copyin()`	29	1	29
ack block `copyout()`	23	1	23
other			2,018
total			44,847

Table 5.4: Profile of a 64-KB `GET_MESSAGE`

5.4. Host Load

The above profiles can be used to show how much host load is incurred by communication processing tasks. The total time necessary for a `SEND_MESSAGE` call is 6.1 ms for a one-byte ASDU and 44.8 ms for a 64-KB ASDU. However, 3.9 ms of the one-byte operation and 23.5 ms of the 64-KB operation are spent waiting on command completion and are therefore free for other host processing. Thus the one-byte call effectively consumes only 2.2 ms, and the 64-KB call consumes only 21 ms of actual host processor time. Similarly, the total time necessary for a `GET_MESSAGE` call is 6.1 ms for a one-byte ASDU and 44.8 ms for a 64-KB ASDU. However, 4.0 ms of the one-byte operation and 17.5 ms of the 64-KB operation are spent waiting on command completion and are therefore free for other host processing. Thus the one-byte call effectively consumes only 2.1 ms, and the 64-KB call consumes only 27.3 ms of host CPU time. Furthermore, only one host-processor interrupt is generated for a either a `SEND_MESSAGE` or `GET_MESSAGE` call of any ASDU size.

5.5. Performance Bottlenecks

As can be seen from the profile information above, different components dominate the execution time for a short ASDUs and long ASDUs. This is due to the fact that some sources of overhead are constant for any sending or receiving operation, regardless of ASDU size, whereas other components of execution overhead are dependent upon ASDU size. In this section we identify and describe these sources of overhead, indicating to what degree they act as bottlenecks in our design.

5.5.1. ASDU Length-Independent Overhead

Our most significant ASDU length-independent overhead results from the operating system kernel architecture of the DTC-2 host. Our UNIX character device driver invokes two standard device driver utility routines called `physio()` and `iodone()` as a part of the execution of the `read()` and `write()` driver entry point routines. The `physio()` routine performs several services for the device driver. It locks into physical memory the virtual pages of the memory buffer (in our case a control or acknowledgment block) referred to by the pointer in the `write()` or `read()`, segments the buffer into fragments that do not violate the size constraints of the underlying device or consume too many system resources, and ensures that the driver has exclusive access to the memory of the buffer. The `iodone()` routine acts to release the exclusive access to the block acquired by the `physio()` operation [46]. These two routines, as can be seen from the profiles, form the largest proportion of the host's execution time spent on processing sending and receiving operations on short ASDUs.

In an effort to ensure security for multiple user processes, UNIX provides protection at the cost of performance through the use of the kernel. We chose to embrace this philosophy in our design as well. As discussed in chapter four, to perform any hardware-dependent operation, our API software must do a system call. Such calls are the second most significant contributors to ASDU-length independent overhead. The raw system call

mechanism itself is only somewhat expensive, i.e., some calls take as little as 22 μs; however the `read()` and `write()` calls also pass through the UNIX I/O system. As a result, they incur overhead from such operations as mapping their UNIX file descriptor to the appropriate device driver entry point and managing buffer-oriented kernel data structures.

As stated in chapter four, our architecture makes use of UNIX signals as a user-level interrupt mechanism indicating the presence of pending acknowledgment blocks. The arrival of such a signal preempts the execution of the application process' next CPU instruction and initiates a signal handling routine. When the signal handler returns, the system resumes the process' normal execution at the next instruction. This mechanism requires processor context to be both saved and restored and therefore takes an appreciable amount of time.

At the user level, a notable amount of time is spent disabling and enabling signals. The reason for such operations is to allow reenttrancy; we disable signals to provide exclusive access to certain global data structures used in the API implementation. There are two basic sources of contention for global API data structures. One is from other Ada tasks in the application process, since they may be preemptively scheduled via signals and may then invoke API primitives. The other source of contention is signal handling routines since they run asynchronously and may contain code that makes API calls. In particular, the `SIGIO` handler, which handles user-notification of pending status is a source of contention for global data structures. This handler is an integral part of the API implementation; it does not make API calls, but it does manipulate many global API data structures. Disabling signals during the use of such data structures wards off the ill effects of other tasks and signal handlers. Once exclusive access to the data structures is no longer needed, a call to re-enable signals becomes necessary. The relatively long duration of these otherwise simple signal-oriented operations is due to the fact that they require the invocation of the `sigblock()` or `sigsetmask()` system calls.

Two minor sources of notable length-independent host overhead are the time required to write a control block to the TN queue and to read an acknowledgment block from the FN queue as discussed in section 4.5.3.2. In our architecture, the `copyin()` kernel utility function is utilized for the write of a control block. It copies, with programmed I/O, a region of memory residing in user virtual address space into kernel virtual address space. In the case of this `copyin()`, the kernel region is a slot in the TN queue located in protocol processor's on-board memory. Thus the copy occurs over the DTC-2's VMEbus. In a similar fashion, the `copyout()` function copies an acknowledgment block out of a slot in the FN queue, which is mapped into kernel virtual address space, to memory in the user-level address space. The duration of these copies are dependent upon the length of the control and acknowledgment blocks, which, in our case, are 104 and 64 bytes, respectively.

5.5.2. ASDU Length-Dependent Overhead

The uniquely significant components of host execution time for large ASDUs are those required to manage the large quantities of data. The overhead of such operations grows with ASDU length and is expectedly evident in the data buffer allocation and the data `copyin()` and `copyout()` components of the profiles for large-ASDU operations. First we note that the time spent on the allocation of buffer space on the protocol processor board becomes noticeable when the data buffers are large. However, this time is insignificant when compared with that of the data copies.

The host-driven data copies are worthy of much investigation. The data `copyin()` transfers an ASDU into a region of kernel virtual address space from user virtual address space. The kernel region refers to protocol processor buffer, and the copy is performed by the host CPU using write transactions on the VMEbus. Similarly, a data `copyout()` copies an ASDU out of a protocol processor buffer in kernel virtual address space into memory in user address space. Here the copy is performed by the host CPU using VMEbus read transactions.

In order to further understand the effect of these data copies on the performance our architecture, we present a more detailed analysis. As one would expect, time average measurement of `copyin()` and `copyout()` indicates that the increase in the latency of these operations scales linearly with the transfer size. In addition to adding latency to our off-host communications architecture, the `copyin()` and `copyout()` operations also constrain our throughput. In measurements of the peak transfer rates of `copyin()` and `copyout()` over the VMEbus and between host and protocol processor memory, the `copyin()` operation has a throughput of 28.6 Mbps and the `copyout()` has a throughput of 21.9 Mbps. We believe that the reason the `copyin()` throughput is the higher of the two is that bus read transactions are often inherently faster than bus writes as discussed in section 2.4.2.3.

To illustrate the throughput-limiting effects of the data `copyin()` and `copyout()` operations on our communications architecture, we operated our system with them removed from the code. The copies of control and acknowledgment blocks between the host and protocol processor were left intact, however. As shown in Figure 5.3, the resulting user-level throughput was over twice as high in some cases. The user-level throughput shown here was measured in the same manner as the other user-level measurements in section 5.2.3.

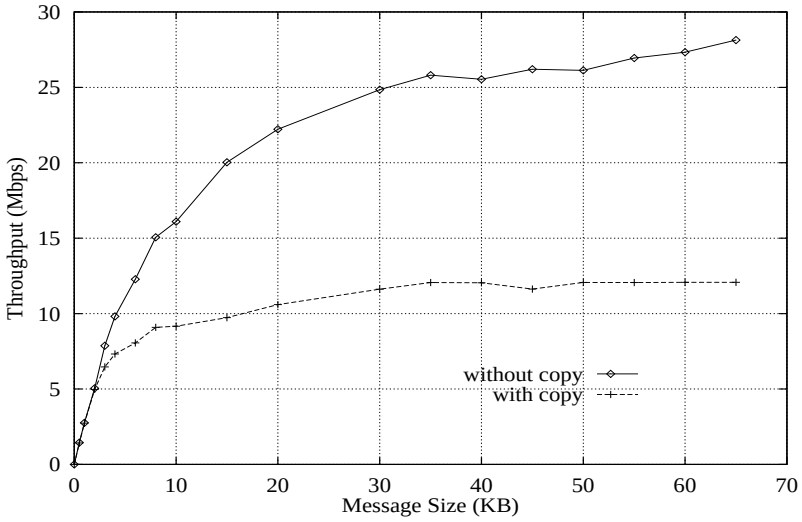


Figure 5.3: Data Copy Effects upon Throughput vs. ASDU Size

Given these measurements, one might hastily blame the VMEbus for the relatively poor throughput performance of our architecture. While this is partially true, it is not the case that the VMEbus is alone responsible for the cap on throughput. Although our programmed I/O VMEbus copies have a low throughput, other VMEbus transfers in our architecture are much more respectable. For example, as shown in section 5.2, in spite of the VMEbus we have achieved MAC throughput far higher than the 21.9 Mbps limit of the data `copyout()`. In fact, we have observed block mode DMA transfers of 64 Mbps from our protocol processor's 70-ns DRAM to the transmit FIFO register on the network interface. Hence there is more overhead in the `copyin()` and `copyout()` than simply that of the VMEbus itself.

The performance of the VMEbus can be affected by several factors. These include whether the transfer is performed using programmed I/O or a DMA controller. With programmed I/O, the speed of the CPU may affect the speed of data transfer. In addition, it may be the case that the CPU can not employ block mode transfers, but the DMA controller on

the network interface can. Another important issue affecting the bus transfer rate is the access or cycle time of the memory or registers on the slave side of the bus transaction. Some measurements of VMEbus throughput [5] have indicated that, for block mode transfers involving infinitely fast memory, the VMEbus can support a throughput of 223.2 Mbps; however, with block mode and a memory having a 150-ns access time, peak VMEbus throughput drops to only 108.8 Mbps. When using the same memory, but no block mode, throughput drops further to 103.2. Mbps. Hence, our low throughput rates argue for more than simply a faster bus; they also argue for a combination of such features as burst mode bus transactions, a high-speed DMA controller, and memory and registers with a low access time.

Finally we turn to the issue of the bus contention resulting from data transfers across the VMEbus. In our architecture, such transfers occur between the host and protocol processor and the protocol processor and network interface; however, only a single such device can master the bus at once. The bus contention due to this restriction acts to block the transfers of such devices, thus hindering their parallel operation. If full parallelism between the host and protocol processor were achieved, one would imagine that our user-level throughput would be somewhere around the minimum of the transport protocol throughput and programmed I/O transfer rate of the host. Unfortunately this is not the case. For example, the peak throughput of our transport protocol is about 24 Mbps, and the peak throughput of our programmed I/O transfers is about 22 Mbps; however, our peak throughput in asynchronous mode is only about 12 Mbps. In fact, we observe in Figure 5.4 that our user-level throughput using synchronous API calls (those exhibiting essentially no host and protocol-processor parallelism by definition) is not particularly lower than the throughput in the asynchronous API mode (where host and protocol-processor parallelism should be present). This suggests that we obtain little parallelism between the host and the protocol processor. These results point to problems of bus contention and thus argue for placing the network interface on the same board as the protocol processor.

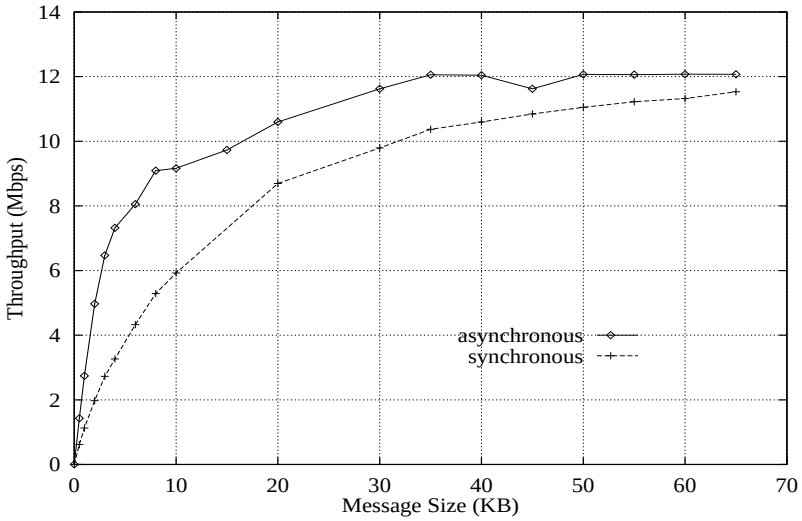


Figure 5.4: API Mode Effects upon Throughput vs. ASDU Size

6 A Simple Model to Predict Performance

6.1. Motivation

In light of the off-host communications architecture discussed in chapter four and its performance presented in chapter five, a fundamental question about our implementation is “How well did we exploit our hardware and software architecture in order to maximize user-level performance and minimize host load?” To answer this question, we wish to evaluate the performance intrinsic to architectural design itself to measure the limits of its performance independent of our implementation of the design. With this knowledge we might discover that the process of making optimizations to the implementation within the framework of our basic architectural design could provide fruitful performance gains. On the other hand, the evaluation may show that we have essentially reached the inherent constraints of our chosen hardware and software architecture. In this case, no amount of optimization within the framework of the architecture would expand its performance beyond these constraints. In such a situation, a new set of fundamental architectural design choices would be in order if we desire to obtain better performance.

In addition to questions which evaluate the performance of an implemented architecture, we may also desire to predict the possible performance of a proposed design. Thus, a question for developers of off-host communications architectures is “How can we predict the possible performance of a proposed off-host architecture before its implementation has begun?” The answer to such a question would be of great interest during the design stages of a system and before the procurement or production of its hardware and operating system.

In this chapter we develop an architecture-based performance model for off-host communications architectures which can be used to provide answers to the above questions of performance evaluation and prediction. This model is based on the general architectural design decisions discussed in chapter two; thus, it can be applied during the design stages of a system before its implementation has begun.

6.2. Background

Performance estimations have been used by others to predict the performance of their designs. In a previous work on off-host protocol processing, Kanakia and Cheriton estimate the performance of their proposed Network Adapter Board [27]. Their analysis takes into account such factors as bus transfer time, network transfer time, and memory access time in order to predict the request-response delay of their device. The approach is quite informal, and it takes into account factors above and below the level of their transport protocol (VMTP).

Our approach differs from Kanakia and Cheriton's in two main areas. First, our performance predictions result from the evaluation of simple, yet formal, mathematical expressions. Second, the transport protocol is treated as a "black box," making its performance characteristics parameters of the model[1]. That is, the transport protocol processor is analyzed in isolation, outside the context of the host CPU. Its performance parameters are determined and then used in the model to predict the performance that will be visible at the user level in the completed system. Rather than concerning itself with the modelling of protocol performance, our model deals mostly with the overhead introduced by the means of communication between the components of a particular off-host communications architecture.

6.3. Goals of the Model

Our performance model has as its goals *simplicity* and *practicality.* With our model there is no need to acquire the probability distributions of any input parameters, as would be required of a queueing theory model. Rather, one need only use readily-available, time-average measurements or vendor-provided specifications[2]. With minimal effort, one can predict the end-to-end latency and throughput seen by application processes at the user

1. Since the transport protocol performance encapsulates that of lower-layer protocols, the performance of such protocols is not considered explicitly in the model.
2. The issue of obtaining these quantities is dealt with in the appendices.

level of an off-host communications architecture. In addition, one may also obtain an estimate of the amount of host processing time required for data transfer operations.

Such predictions are practical in that they are of interest to the developers of systems making use of off-host protocol processing. System architects wish to know to what degree they either will capitalize or have capitalized on the possible benefits of off-host architectures presented in chapter one. If users of the model find it lacking in a particular area, they may extend it to meet their needs using the general methodology of the model; however, such extension occurs at the cost of increased complexity.

6.4. Model Overview

Without going into the analysis of a specific architecture, this section presents an overview of the input components of the model and how they are formulated into output predictions. In addition, we describe the types of communications operations considered in our model.

Figure 6.1 presents a very general illustration of the functions of the model. Input parameters based upon a specific architectural design and the performance of the chosen protocol processor are supplied to the model. In return, the model predicts the user-level performance of the architecture including end-to-end latency, throughput, and host load.

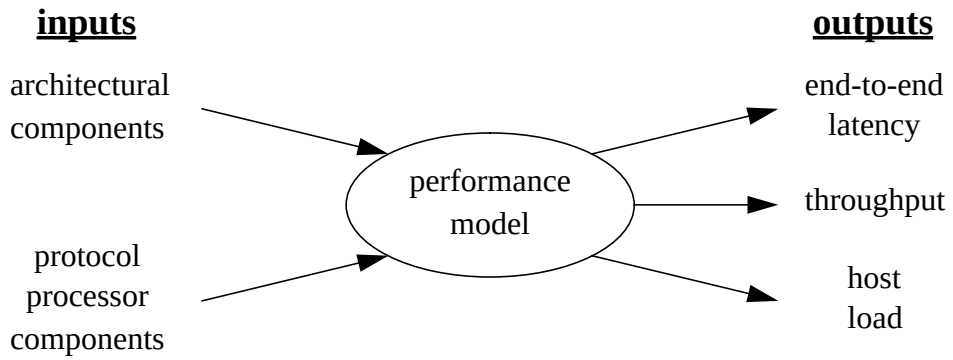


Figure 6.1: Overview of the Performance Model

6.4.1. Input Components

Here we provide a general overview of the inputs to the architecture-based performance model. Regardless of the specific input parameters used, the input components of the model can be divided into two general classes: architectural components and protocol processor components. The architectural components represent those performance factors which are strictly a function of the host computer system and the mechanisms which allow it to communicate and synchronize with the protocol processor. They represent the effects of the system designer's decisions on the issues discussed in chapter two, except for the design of the protocol processor itself. Essentially, these architectural parameters characterize the performance of the data and control paths through the communications architecture.

Other inputs to the model characterize the performance of the particular protocol processor used in the off-host communications architecture. In our model, the protocol processor performance is deliberately distinguished from that of the surrounding architectural components. Due to the complex nature of transport protocols, e.g., their flow and error control, it is difficult to analytically predict transport performance. To keep our performance model simple, we choose to avoid such complexity by treating the protocol processor as a black box described by a set of readily-obtainable performance parameters. These parameters may be easily determined from the specifications of the protocol processor vendor or from the empirical study of the device.

6.4.2. Determination of Outputs

The general methodology of the performance model is as follows. The analysis integrates the input parameters to provide sums of terms which account for the delay introduced by components of the architecture's data path. In addition, the model takes into account the processor parallelism which causes overlap in components of host and protocol processor execution. This parallelism causes certain terms to drop out of the analysis.

Before we explain the model's outputs, we first introduce some terminology for two quantities used in the analysis, *delay* and *period*. In our model, *delay* measures the time between the initiation of a transmitting operation at one endpoint and the completion of the corresponding receiving operation at a peer endpoint. In contrast, the *period* of an operation is the time between the beginning of the operation and the *local indication* of its completion. The period is so named because it represents the minimum time between two consecutive operations such that there is no overlap in their execution. This basic quantity can subsequently be scaled to take any parallelism in operation execution into account.

The model includes formulas for several quantities; some values are intermediate results used to predict outputs, and others are the output predictions themselves. First we must note that all of the model's predicted outputs, as well as many terms, are dependent upon the ASDU length, represented here by the symbol l. The three most fundamental quantities of the model are the user-level delay, $\delta(l)$, the user-level period for a transmitting operation, $T_t(l)$, and the user-level period for a receiving operation, $T_r(l)$. The delay is a final output, predicting the user-level end-to-end latency of the off-host communications architecture (in units of seconds). The periods are intermediate quantities used to predict two more output quantities (in units of bytes per second), the user-level throughput for synchronous transfers, $\tau_s(l)$, and the user-level throughput for asynchronous transfers, $\tau_a(l)$. In addition, one may predict the two final output quantities, the host processing time (in units of seconds) required to transmit an ASDU, $H_t(l)$, and to receive an ASDU, $H_r(l)$.

6.4.3. Communication Operations Considered

Since we intend to describe only the throughput, latency and host load of an off-host communications protocol architecture, we consider only a minimal set of communications primitives, those for connection-oriented data transmission and reception. The rationale for this decision is as follows. Foremost, the decision to limit the number of operations studied keeps our model simple. With this in mind, we chose to model the communications opera-

tions of greatest interest to the developers of off-host architectures. We think that connection-oriented data transmission and reception operations meet this criterion.

Other communications operations could be studied. For example, it is usually the case that communications operations exist for such tasks as connection management and option manipulation. While these operations are very important, they largely serve to test the performance of a specific transport protocol rather than that of an off-host communications architecture. In addition, transmitting and receiving operations are often present in two forms, connection-oriented and connectionless. To model both types of operations would needlessly complicate our model. For this reason we choose only to model the connection-oriented transmitting and receiving operations.

It is recognized that, at the cost of greater complexity, the modeling of the other operations mentioned above could be useful. With this in mind, the analysis of a larger set of communications operations is considered in section 6.10.3.

6.5. Assumed Class of Architectural Design

Now that we have described our model in general, we target it toward a specific class of off-host communications protocol architectures. The protocol processor parameters of the model are quite general, and are treated independently of the surrounding off-host communications architecture. As a result, they are independent of the architectural design class, so they are not considered here. This is not the case, however, with the architectural parameters.

Given a particular architectural design, several specific components act as the members of the general class of architectural components in our model. In contrast with the protocol processor parameters, these specific components of an architecture are somewhat dependent upon the fundamental design choices discussed in chapter two. That is, within the general class of architectural component parameters, we must choose specific components which target the model to a particular design class of off-host communications archi-

tecture. For this reason, the resulting analysis is somewhat architecture-specific; however, the methodology employed in arriving at the results is quite general. To this effect, we argue that, although here we target our model toward a specific architecture, the general framework of the model can be applied to other architectural classes as well. This is discussed further in section 6.10.1.

The specific class of architecture we consider here is that which follows the design of the off-host communications architecture for SAFENET described in chapter four. To review, this architecture has a user level, a protected kernel level, a protocol processor level, and a network interface level. The user and kernel levels reside on the host CPU board, a module which resides on a backplane bus along with separate protocol processor and network interface modules.

Between the architectural levels, we employ several communication and synchronization mechanisms. The interface between the user and kernel levels is essentially via system calls for command and status, shared memory for data, and interrupts for notification. Between the kernel and protocol processor layers, communication is via command and status queues and buffers residing in shared memory. This information is transferred to and from these data structures through the use of host-driven programmed I/O over the bus. The protocol processor interrupts the host CPU for notification of pending status. Using these mechanisms, three types of information flow between the host and protocol processor; these are TSDU data, commands, and status. The only facet of the communication between the protocol processor and network interface that is relevant to our model is the fact that such communication occurs over the backplane, thereby generating bus traffic and possibly causing contention.

6.6. Input Parameters for our Architectural Class

Recall that all of our input parameters may be classified into two groups, those specific to a particular architectural class and those that characterize the protocol processor.

The architectural parameters of the model and their symbols are listed in Table 6.1. The choice of the particular parameters was determined in part by the profiles in chapter five. Only those delay components that were found to be significant are included in the table. This is why the time for the protocol processor to interrupt the host is not included, while the time for the user-level interrupt of the application process is included.

Symbol	Meaning
δ_w	round-trip delay of the `write()` driver entry point
δ_r	round-trip delay of the `read()` driver entry point
δ_i	round-trip user-level interrupt delivery time
l_c	length in bytes of a control block
l_a	length in bytes of an acknowledgment block
ρ_w	per-byte time required for the host to write data or control information to the protocol processor
ρ_r	per-byte time required for the host to read data or acknowledgments from the protocol processor

Table 6.1: Architectural Input Parameters

For the model to be useful during the development stage of an architecture, it is important not to include any components which are dependent upon a specific software implementation. For this reason there are no parameters in the table for operations such as the operation to disable signals, even though it is significant in the profile. The inclusion of such components is discussed in the context of evaluating the existing implementations of architectures in section 6.10.2.

The protocol processor parameters of the model are given in Table 6.2. In contrast to the architecture-dependent set of parameters, those that characterize the surrounding architecture, the same set of protocol processor parameters may be used for any particular protocol processor. Furthermore, such parameters are not in any way dependent upon the architectural class. The specific choice of parameters was not determined by examining the profiles; rather, the relevant quantities were chosen because they are easy to obtain and fit simply into the model's derivations.

Symbol	Meaning
δ_0	time required for the protocol processor to transmit a minimum-length TSDU end-to-end
ρ_δ	per-byte time required for the protocol processor to transmit a TSDU end-to-end
T_0	period of the protocol processor in transmitting a minimum-length TSDU
ρ_T	per-byte addition to the period of the protocol processor in transmitting a TSDU

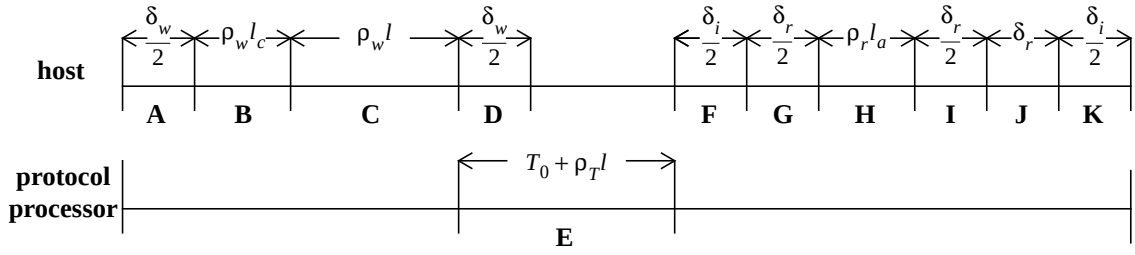
Table 6.2: Protocol Processor Input Parameters

6.7. Performance Derivations

Derivations for user-level period, throughput, and delay, as well as host load are presented below. All derivations are based on summations of time values; therefore, in the following analysis, the ρ-quantities (in units of seconds/byte) are multiplied by the *l*-quantities (in units of bytes) to produce elapsed times.

6.7.1. Period Derivation

The calculation of the user-level periods for transmission and reception proceeds in this section. These results will later be used to calculate throughput and, to some extent, host load. To estimate the user-level period for transmission, we undergo the following analysis. Processor execution time lines for a transmitting operation are given in Figure 6.2. Note that δ_w, δ_r, and δ_i are defined above as round-trip times rather than one-way times because round-trip times are easier to measure in practice. The one-way time of each is then estimated as simply half the round-trip time. The δ_r term at location **J** is subtle. Because interrupts do not queue, a single interrupt may signal the arrival of more than one protocol-processor acknowledgment. The δ_r term represents a second read of the protocol processor's status to determine if another acknowledgment is pending. Given these time lines, we may sum their serial components to obtain the user-level period for a transmitting operation as follows.



A: time to enter the device driver `write()` routine
B: time to copy the transmit command to the protocol processor
C: time to copy the user data to the protocol processor
D: time to return from the device driver `write()` routine
E: protocol processing time required to service the request
F: time to initiate the user-level interrupt handler routine
G: time to enter the device driver `read()` routine
H: time to read the acknowledgment from the protocol processor
I: time to return from the device driver `read()` routine
J: time to enter and return from the device driver `read()` routine
K: time to return from the user-level interrupt routine

Figure 6.2: Delay Components of the User-Level Period for Transmission

$$T_t(l) = \frac{\delta_w}{2} + \rho_w l_c + \rho_w l + \max(\frac{\delta_w}{2}, T_0 + \rho_T l) + \frac{\delta_i}{2} + \frac{\delta_r}{2} + \rho_r l_a + \frac{\delta_r}{2} + \delta_r + \frac{\delta_i}{2}$$

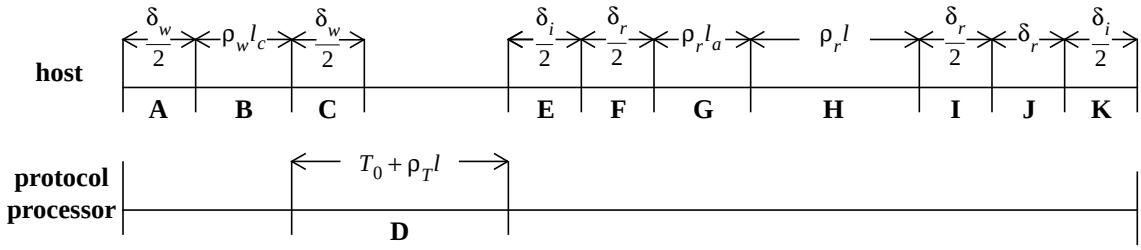
$$= (\rho_w + \rho_T)\, l + \frac{\delta_w}{2} + T_0 + \delta_i + 2\delta_r + \rho_w l_c + \rho_r l_a$$

We base the second step of the derivation on the assumption that $\delta_w/2 \le T_0 + \rho_T l$, i.e., **D** is dominated by **E**, which should typically be the case. This is an example of where processor parallelism causes a serial delay term to drop out.

The analysis of the user-level period for reception is similar and proceeds as illustrated in Figure 6.3. As with the transmitting operation, we make the assumption that $\delta_w/2 \le T_0 + \rho_T l$; therefore, the **C** term drops out of the analysis, justifying the second step of the derivation. The period for a receiving operation is derived as follows:

$$T_r(l) = \frac{\delta_w}{2} + \rho_w l_c + \max(\frac{\delta_w}{2}, T_0 + \rho_T l) + \frac{\delta_i}{2} + \frac{\delta_r}{2} + \rho_r l_a + \rho_r l + \frac{\delta_r}{2} + \delta_r + \frac{\delta_i}{2}$$

$$= (\rho_r + \rho_T)\, l + \frac{\delta_w}{2} + T_0 + \delta_i + 2\delta_r + \rho_w l_c + \rho_r l_a$$



A: time to enter the device driver `write()` routine
B: time to copy the receive command to the protocol processor
C: time to return from the device driver `write()` routine
D: protocol processing time required to service the request
E: time to initiate the user-level interrupt handler routine
F: time to enter the device driver `read()` routine
G: time to read the acknowledgment from the protocol processor
H: time to copy the user data from the protocol processor
I: time to return from the device driver `read()` routine
J: time to enter and return from the device driver `read()` routine
K: time to return from the user-level interrupt routine

Figure 6.3: Delay Components of the User-Level Period for Reception

6.7.2. Throughput Derivation

In this section we calculate the user-level throughput for both synchronous and asynchronous data transfers. To calculate the throughput of the individual transmitting or receiving components of our architecture, we may simply divide their periods into l, the ASDU length. The calculation of the synchronous user-level throughput is a function of the throughputs of the individual transmitting and receiving components. To derive it we must note that, in steady-state operation, the throughput of the communications architecture can be no greater than that of its slowest component; therefore, from $T_t(l)$ and $T_r(l)$ we obtain the synchronous user-level throughput as follows:

$$\tau_s(l) = \frac{l}{\max(T_t(l), T_r(l))}.$$

The derivation of the asynchronous user-level throughput, $\tau_a(l)$, is more complex due to the fact that we must take into account both the parallelism of the host and protocol processor and the shared resource of the backplane bus. This derivation is based upon the

following assumptions about the steady-state operation of the communications architecture when asynchronous API calls are continuously performed:

- The host keeps the command queue to the protocol processor full.
- The protocol processor is always either busy or blocking on bus access.
- An insignificant portion of T_0, the protocol processor period of a minimum-length TSDU, is spent on bus accesses.
- The TSDU-length-dependent portion of the protocol processor period, $\rho_T l$, is dominated by the time for bus transactions.

For the architecture described in chapter four, these assumptions have been verified to hold *for large ASDUs* through the direct examination of VMEbus traffic. Taken together with the processor parallelism due to asynchronous operation, the assumptions imply that all terms of $T_t(l)$ and $T_r(l)$ that do not require access to the backplane bus drop out of the analysis. These terms are $\delta_w/2$, T_0, δ_i, and $2\delta_r$; therefore, for sufficiently large l we have:

$$\tau_a(l) = \frac{l}{\max(T_t(l), T_r(l)) - (\frac{\delta_w}{2} + T_0 + \delta_i + 2\delta_r)}.$$

6.7.3. Host Load Derivation

In order to characterize the host load incurred when using an off-host communication architecture we provide estimates of the amount of host CPU time required to perform transmitting and receiving operations. The minimum host processing time required to transmit an ASDU can be derived from Figure 6.2. By summing the delay components on the host time line, i.e., components **A**-**D** and **F**-**K**, we arrive at the following.

$$\begin{aligned} H_t(l) &= \frac{\delta_w}{2} + \rho_w l_c + \rho_w l + \frac{\delta_w}{2} + \frac{\delta_i}{2} + \frac{\delta_r}{2} + \rho_r l_a + \frac{\delta_r}{2} + \delta_r + \frac{\delta_i}{2} \\ &= (l_c + l)\,\rho_w + \delta_w + \delta_i + 2\delta_r + \rho_r l_a \end{aligned}$$

The minimum host processing time required to transmit an ASDU can be derived in a similar fashion from Figure 6.3. Summing the delay components on the host time line, i.e., components **A**-**C** and **E**-**K**, we arrive at the following.

$$H_r(l) = \frac{\delta_w}{2} + \rho_w l_c + \frac{\delta_w}{2} + \frac{\delta_i}{2} + \frac{\delta_r}{2} + \rho_r l_a + \rho_r l + \frac{\delta_r}{2} + \delta_r + \frac{\delta_i}{2}$$

$$= (l_a + l)\,\rho_r + \delta_w + \rho_w l_c + \delta_i + 2\delta_r$$

6.7.4. Delay Derivation

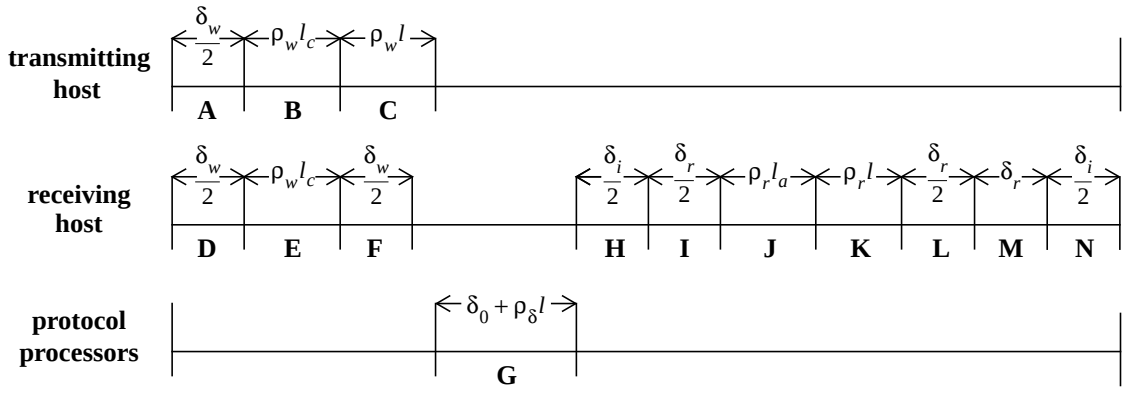
The expression for $\delta(l)$ is based on the analysis illustrated in Figure 6.4. In this analysis, we calculate the user-level delay by studying the parallel execution of the transmitting host, receiving host, and their protocol processors. We arbitrarily assume that the transmitting and receiving operations start simultaneously. Before the protocol processors can perform component **G** successfully, both the transmitter and receiver must issue their requests to their local protocol processors; thus, components **A**-**E** must all complete execution in order to begin component **G**. Thus they are, in effect, in series with the execution of **G**.

Because components **A**, **B**, and **C** on the transmitter and **D** and **E** on the receiver share no resources, their execution proceeds in parallel. Furthermore, the execution of **A** and **B** coincides exactly with that of **D** and **E**; thus, we need only include delay terms for the **A** and **B** components in our expression for $\delta(l)$.

We also choose to exclude the **F** component from our sum since this component is in parallel with components **C** and **G**, and it is almost certainly the case that $\rho_w l + \delta_0 + \rho_\delta l \geq \delta_w / 2$, i.e., the combined execution of components **C** and **G** takes longer than that of **F**. Based on these arguments, the resulting expression is calculated as follows.

$$\delta(l) = \frac{\delta_w}{2} + \rho_w l_c + \rho_w l + (\delta_0 + \rho_\delta l) + \frac{\delta_i}{2} + \frac{\delta_r}{2} + \rho_r l_a + \rho_r l + \frac{\delta_r}{2} + \delta_r + \frac{\delta_i}{2}$$

$$= (\rho_w + \rho_\delta + \rho_r)\, l + \frac{\delta_w}{2} + \delta_0 + \delta_i + 2\delta_r + \rho_w l_c + \rho_r l_a$$



A: time to enter the device driver `write()` routine
B: time to copy the transmit command to the local protocol processor
C: time to copy the user data to the protocol processor
D: time to enter the device driver `write()` routine
E: time to copy the receive command to the local protocol processor
F: time to return from the device driver `write()` routine
G: protocol processing time required to service the request
H: time to initiate the user-level interrupt handler routine
I: time to enter the device driver `read()` routine
J: time to read the acknowledgment from the protocol processor
K: time to read the copy the user data from the local protocol processor
L: time to return from the device driver `read()` routine
M: time to enter and return from the device driver `read()` routine
N: time to return from the user-level interrupt routine

Figure 6.4: Delay Components of the User-Level Delay

6.8. Model Predictions for our Architecture

In this section we apply the model to our communications architecture for SAFENET, producing estimates based on its input parameter values. These predictions are later validated in section 6.9.

6.8.1. Input Parameter Values

For our architecture, the inputs to the model have the values listed in Table 6.3. Those for the architectural components were gathered using the time average measurement techniques described in Appendix A. The protocol processor parameters were determined from the performance plots of transport protocol performance given in chapter five. The method of their extrapolation from such plots is described in Appendix B.

Parameter	Value
δ_w	392 µs
δ_r	382 µs
δ_i	143 µs
l_c	104 bytes
l_a	64 bytes
ρ_w	279.45 ns/byte
ρ_r	364.8 ns/byte
δ_0	3.112 ms
ρ_δ	363.97 ns/byte
T_0	2.062 ms
ρ_T	295.82 ns/byte

Table 6.3: Input Parameter Values for our SAFENET Implementation

6.8.2. Throughput and Latency Predictions

Given this input, the model predicts the end-to-end latency shown in Figure 6.5, the synchronous throughput shown in Figure 6.6, and the asynchronous throughput shown in Figure 6.7. Each prediction is as a function of ASDU length.

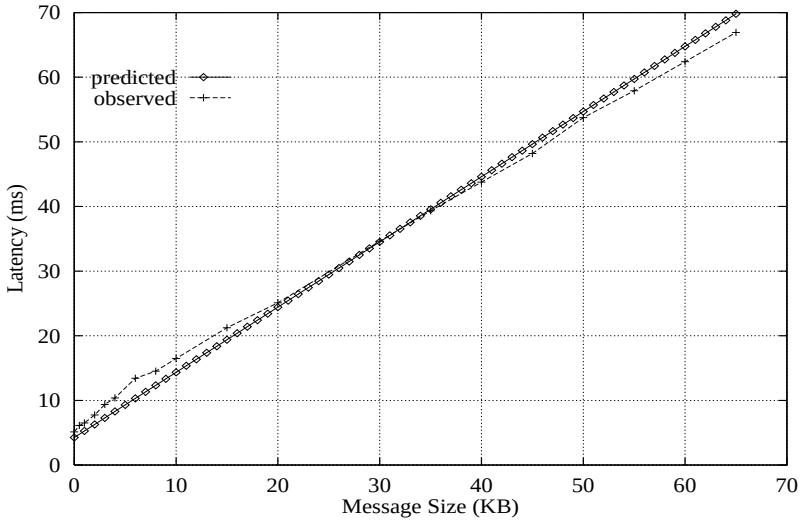


Figure 6.5: End-to-End Latency vs. ASDU Size

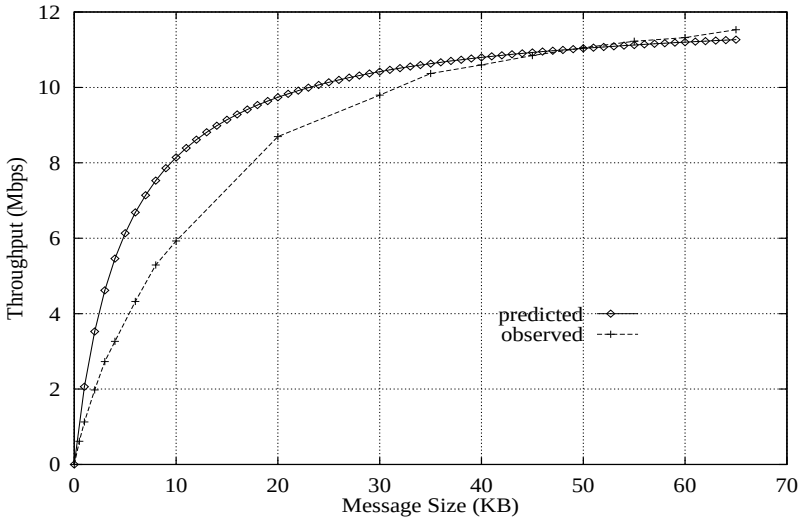


Figure 6.6: Synchronous Throughput vs. ASDU Size

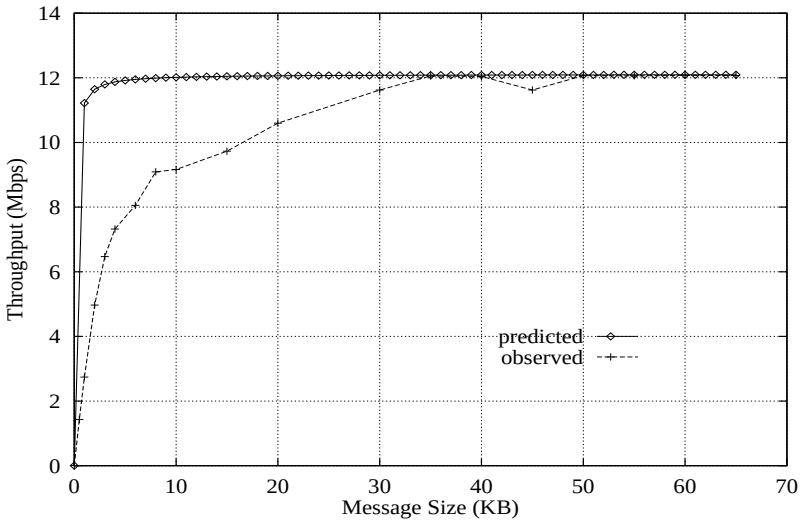


Figure 6.7: Asynchronous Throughput vs. ASDU Size

6.8.3. Host Load Predictions

Using the host load formulas derived in section 6.7.3, the model predicts the minimum host processing time to transmit an ASDU to be 1.4 ms for a one-byte ASDU and 19.5 ms for a 64-KB ASDU. To receive an ASDU, the model predicts a time of 1.4 ms for a one-byte ASDU and 25.1 ms for a 64-KB ASDU.

6.9. Comparison with Empirical Results

In order to validate the predictions of our model, we compare them below with actual performance observations from chapter five.

6.9.1. Validation of Throughput and Latency Predictions

From examination of Figure 6.5 we can see that the end-to-end latency prediction is a good estimate of the observed user-level end-to-end latency. Figure 6.6 shows that the synchronous user-level throughput predictions of the model are relatively accurate for large ASDUs, but overestimate the throughput for short ASDUs. This discrepancy indicates that there is more delay overhead in the data path than the terms of $T_s(l)$ and $T_r(l)$ take into account. As show in Figure 6.7, the asynchronous user-level throughput predictions are quite good for large ASDU sizes; however they are totally inadequate for short messages. This was to be expected, given the fact that our analysis was predicated on assumptions which hold only for large ASDUs.

6.9.2. Validation of Host Load Predictions

In chapter five we observe that the host processing time required for transmission is 2.2 ms for a one-byte ASDU and 21 ms for a 64-KB ASDU. A comparison of the observed host processing time to perform a transmission with the predictions in the previous section indicate a 36% difference for a one-byte ASDU and a 7.1% difference for a 64-KB ASDU. We also observe in chapter five that the host load for a reception is 2.1 ms for a one-byte ASDU and 27.3 ms for a 64-KB ASDU. In comparison with the host load predictions for reception in the previous section, there is a 33% difference for a one-byte ASDU and a 8.1% difference for a 64-KB ASDU.

6.10. Possible Extensions to the Model

The formulae presented above are limited in application to designs which belong to our specific architectural class. In addition we made many simplifying assumptions in order

to make our analysis tractable. For example, we avoided a queuing theory approach in order to simplify the mathematics. In spite of these decisions, our general model may be extended in various ways to handle other architectures or to provide greater accuracy at the cost of higher complexity. This section outlines several possible avenues of model extension.

6.10.1. Support for Other Architectural Classes

Although it is applied above to the class of architectural design discussed in section 6.5, the general methodology of our model can be used with other architectures as well. For example, given an architecture in which device driver does not reside in a protected kernel or in which user processes memory map the protocol processor, system calls may not be necessary. In such cases, the system call components of our model, δ_w and δ_r, may simply be set to zero. In another design, suppose that DMA transfers are used to transfer TSDUs to the protocol processor; in this case, the bus transfer times (those terms involving a ρ_w or ρ_r) may be omitted from the derivations of host load. As yet another example, suppose the buffering of data on the protocol processor is omitted. With this scenario, the cost of data copies to and from the protocol processor can be dropped from the entire analysis. Of course, the data is still copied at some point; however, the cost of such a copy is subsumed within the protocol processor parameters.

6.10.2. Implementation-Dependent Components

The accuracy of the model can be increased by taking into account implementation-dependent input parameter components. During or after the implementation of the design of a particular off-host communications protocol architecture, knowledge of the overhead of specific implementation components can be added to the model. These components supplement those that describe the performance of the specific architectural class and protocol processor. This approach adds complexity to the model; however, it should also provide more accurate predictions of the performance of the final implementation.

As an example, we consider the profiles in chapter five of our SAFENET architecture. These profiles indicate that the act of disabling and enabling signals is significant. In addition they indicate that, for large messages, the time to allocate buffers becomes appreciable. Noting the significance of these artifacts of our implementation, we could incorporate them into our model as additional parameters in order to obtain greater accuracy, particularly for short ASDUs.

6.10.3. Consideration of More Communication Operations

In order to keep our analysis as simple as possible, we restricted our model to connection-oriented transmit and receive operations. It may be of interest to designers to predict the performance of other communication operations as well. This is possible, using the general methodology of the model; however, it could introduce many new input parameters. For example, if we were to extend our model to predict the user-level end-to-end delay of connection establishment, we would need a new parameter to characterize the delay of the protocol processor for this operation. This is essential for accuracy because connection setup is a relatively expensive operation for many transport protocols.

6.10.4. Use of Queueing Theory

The use of queuing theory was absent from our analysis in order to avoid complex, possibly even intractable, mathematics in the derivation of the model's outputs. In addition, we wished to avoid the difficult task of finding probability distributions for the model's inputs. To avoid such complexity, we chose to use time-average measurements of input parameters and made simplifying assumptions in the model's analysis.

At the cost of complexity, there are certain components of our model which may be much more accurately represented by probability distributions. Thus the additional complexity of a queuing theory approach may provide more accurate results. For example, since protocol processing is a nondeterministic activity, its input parameters would probably be better represented by probability distributions. In addition, the shared use of the

backplane bus depends on statistical utilization factors and various arbitration and release policies. The consideration of these factors may mandate a queuing theory model. Such a model may also be necessary in order to relax certain other assumptions. For example, we assume the use of unloaded hosts in our analysis and do not consider the effects of queuing in the command and status queues to the protocol processor, nor do we consider the effects of masking interrupts on interrupt response time.

7 Conclusions

7.1. Summary

This work has concerned itself with communications architectures in which the transport and network layers of the OSI reference model execute off-host on a protocol processor. In chapter one we pointed out the renaissance in communications architecture design brought about by high-performance LAN technology. We noted that, although these new networks offer both high-performance and new services, a computer system's communications architecture (including hardware, an operating system, and communications protocols) that stands between the network and its ultimate user can mask the benefits of such networks from the user's point of view. In searching for a new architectural approach we explored the issue of protocol placement. We pointed out that there were two fundamental choices concerning where protocols execute, on-host and off-host, and we proceeded to contrast these choices. We then postulated that off-host communications architectures were a promising approach worthy of study. To this end, we identified several possible advantages that off-host architectures could offer both the host computer system and the protocols; however, we also warned of the possible pitfalls of off-host protocol implementation.

In chapter two we outlined the many design issues inherent in off-host communications protocol architectures. These mandated many choices with respect to the channels of communication present in an off-host architecture. Such choices were with respect to the placement of the components implementing the off-host architecture and the mechanisms necessary for communication between components at each level. We also pointed out two philosophical extremes in design, a tightly-integrated design and a loosely-integrated design, and discussed the trade-offs inherent to each.

In chapter three we surveyed the literature containing related work on off-host communications protocol architectures. Some work only resulted in an architectural design and performance estimates, while other work included both a design and actual performance

results. The approaches ranged from tightly-integrated to loosely-integrated designs and thus illustrated the resolution of the many design issues outlined in chapter two.

We presented the design of our own communications protocol architecture in chapter four. To some extent, our design choices were constrained by our host computer and its operating system; still, several design options were still left open to us. Throughout the chapter we illustrated how we resolved such issues. The resulting architecture is very loosely-integrated relative to those discussed in chapter three. It implements the SAFENET lightweight suite of protocols using three VMEbus modules, a DTC-2 host running application processes, a single-board computer acting as an XTP protocol processor, and a network interface. In order to provide multiprogramming security our design makes use of system calls, and, in order to avoid DVMA space restrictions, it uses host-driven programmed I/O and shared TSDU buffers.

In chapter five we analyzed the performance of our implementation of the architecture designed in chapter four. We presented the performance of our implementation at several service levels. In addition we profiled our performance at the user and kernel levels in the host and analyzed the profiles. Using such information and some related work on backplane bus performance, we identified the ASDU length-dependent and length-independent performance bottlenecks in our architecture.

In chapter six we gave the motivation for an architecture-based performance model of off-host communications protocol architectures. We presented the general design of such a performance model having the twin goals of simplicity and practicality. Within this general framework, we targeted the model to the specific class of off-host communications architectures to which our SAFENET architecture belongs. Based on this specific architecture we chose input parameters and derived formulas to predict overall architectural performance. In order to illustrate the use of our model, we applied it to our architecture for SAFENET, displaying the model's performance predictions. Furthermore, we illustrated

the model's validity by comparing these predictions to the performance measured in chapter five.

In this chapter we make conclusions about our off-host communications architecture in light of the results in the previous chapters. First we evaluate the extent to which we achieved the benefits of an off-host communications architecture that were claimed possible in chapter one. Second, we evaluate the extent to which we committed the possible pitfalls of an off-host approach, which are also discussed in chapter one. We then evaluate the simple performance model of chapter six in light of its development, application, and validation. Finally, we identify areas for future work and present general conclusions on the suitability of off-host architectures.

7.2. Evaluated Advantages of our Off-Host Architecture

In this section we evaluate to what extent the architecture described in chapter four provides the possible benefits off-host communications architectures. Some facets of this evaluation require comparison with the behavior an in-host architecture. For this purpose we compare our work with an in-kernel implementation of XTP [47]. This in-kernel approach features a somewhat different API, but the XTP layer and below consists of essentially the same protocol software. Other evaluations consider aspects of our off-host communications architecture which are readily observable from the design presented in chapter four or its analysis covered in chapter five.

7.2.1. Host Benefits

An off-host architecture promises the following host benefits: decreased host CPU load, predictable application processing, reduced and bounded host interrupts, and reduced incident TSDU traffic. Here we consider the extent to which our design provided such benefits. By design, the number of incident TSDUs on our host is equal to the number of receiving operations performed; thus, we can indeed reduce and, in fact, precisely bound incident TSDU traffic by limiting the number of receive operations that the host performs. Similar

benefits also occur with respect to interrupts; each sending or receiving operation incurs exactly one host interrupt. Results from [47] show that 26 interrupts can result for a single 64-KB receiving operation. Thus the off-host architecture incurs 96% fewer interrupts in such cases. Obviously our design is quite successful in acting as a firewall.

The CPU load for transmitting and receiving operations can be characterized (1) relative to the overall time to carry out these operations using our off-host architecture and (2) relative to the host load for the in-kernel implementation. We shall present both such forms of analysis. With respect the total execution time for communication, the host load has been recorded as high at 60.9% of total processing time; this is in the case of a 64-KB `GET_MESSAGE`. The proportion of overall time incurring host load is much smaller for a one-byte `GET_MESSAGE`; the host load due to such a call consists of only 30.4% of total processing time. These results show that the role of the host in an off-host communications architecture can be much larger than one might expect.

To compare the host load of our off-host architecture with that of the in-kernel approach, we compare the load for transmitting operations of both short and long ASDUs. Estimates in [47] show that a 64-KB sending operation takes 66 ms of host CPU time for the in-kernel approach, whereas chapter five shows that a similar call using the off-host architecture requires only 21 ms. Relative to in-kernel operation, the off-host approach reduces host load by a factor of 68%. For a one-byte ASDU, an in-kernel transmitting operation consumes 4.7 ms of host CPU time, whereas a similar call in the off-host architecture consumes 2.2 ms. Here the off-host approach reduces load by only 53%. Clearly we received a major benefit in the reduction to host load provided by our off-host architecture.

We manage to provide another benefit in that the amount of host CPU load in our off-host architecture is deterministic. By design, the host CPU time incurred for communications operations is constant. On the other hand, the host load in-kernel approach can be quite nondeterministic, especially in the face of error or flow control.

7.2.2. Protocol Benefits

Here we discuss the protocol benefits provided by our off-host architecture. The benefits deemed possible are dedicated processing cycles, specialized hardware, an optimized data path to network hardware, and an ideal operating system environment. With our design we gave our protocol, XTP, its own processor; therefore one would trivially conclude that the protocol has dedicated processor cycles. To some extent, however, one could argue that this is not the case since XTP has to share the processor with the FDDI MAC device driver. We refute this by claiming that such processing is actually a part of XTP protocol processing. The MAC driver software includes the fundamental operations necessary to send and receive TPDUs; all true MAC processing is done on the network interface itself.

Essentially, our protocol processor contains no specialized hardware support for protocol processing. The protocol processor is based upon a general-purpose single-board computer. This device was not explicitly designed for protocol processing; however, it does have a few features useful for such processing. These features include an on-board DMA controller and burst mode capability. The DRAMs are in no way specialized for protocol processing; they have only a 70-ns access time, which is approximately the same speed as the host's own memory. In addition the protocol processor's CPU is not specialized. It is clocked at the same rate as the host CPU, but is a CISC processor rather than a RISC processor. It is not clear which type of instruction set design is better for protocol processing.

Another cited advantage of off-host architectures is the possibility of an optimized data path between the protocol processor and network interface; however, our architecture contains no hardware support for such an optimization. To the extent possible, we attempt to optimize this path in software. Our network interface is programmed to employ block mode DMA transfers across the VMEbus between itself and the protocol processor. More traffic crosses the channel between the protocol processor and network interface than that between the host and protocol processor. In our design, the latter path crosses the bus via programmed I/O and without using block mode DMA. Furthermore, such programmed I/

O has been found to have about half the throughput as the DMA between protocol processor and network interface. One can see that we are fortunate to have optimized the path which carries more traffic.

Our choice of operating system for the protocol processor seems to have been a wise one. In measurements of the in-kernel implementation, the UNIX kernel architecture has been shown to severely impact the performance of the protocol through high timer, buffer, tasking, and network interface management overhead. The choice of the lightweight multitasking operating system, pSOS+, and our own custom buffer and network interface management allowed us to perform XTP processing with lower overhead and thus higher performance. As a result, the peak user-level throughput for our off-host architecture was 12.1 Mbps, whereas the peak user-level throughput for the in-kernel approach was only 7.70 Mbps.

7.3. Evaluated Pitfalls of our Off-Host Architecture

In chapter one, we warned of several possible pitfalls which could counter the ability of an off-host protocol implementation to deliver its possible benefits. In this section we evaluate our architecture with respect to these pitfalls. We first consider the pitfalls inherent in both on-host and off-host architectures.

7.3.1. Incurring Operating System Overhead

As with an in-kernel protocol implementation, ours incurs some degree of operating system overhead. As discussed in chapter five, our overhead is due to system calls for writing commands, reading status, and handling, disabling, and enabling signals. The cost of these operations is quite significant for relatively short ASDUs; however, it is quite irrelevant with respect to long ASDUs.

In contrast, the Nectarine API discussed in section 3.4 avoids system call overhead at the cost of providing system security. Our design, on the other hand, chose to incur such overhead in exchange for security from malicious or accidental corruption of the system by

user processes. It is surprising, however, that the vast majority of system call overhead does not occur in the system call mechanism itself, but rather in the `physio()` and `iodone()` kernel utility routines. This suggests that the system call mechanism itself may be less of a factor than one might otherwise believe.

7.3.2. Overtaxing the Host CPU with Protocol Processor Driving

Here we evaluate the extent to which our off-host communications architecture overtaxes the host CPU with tasks associated with operating the protocol processor device. These tasks include the transfer of control, status, and data, the management of shared data structures, the multiplexing and demultiplexing of outgoing commands and incoming status, and the user-level notification of pending status. Any device requires some degree host overhead in order to drive it; our goal was to minimize this overhead so that the host would be off-loaded as much as possible.

As shown in the profiles, the cost of the transfer of control blocks and status blocks to the protocol processor is relatively insignificant, even for short ASDUs, when compared with UNIX overhead. In contrast, data transfer can be quite taxing on the host. For example, with a 64-KB `GET_MESSAGE` operation, the majority of overall processing time (including that of both the host and protocol processor) is spent by the host performing the data `copyout()`. We, as designers, were shocked to find that the programmed I/O transfer across the VMEbus was so time consuming. The throughput of the bus for `copyout()` operations is actually lower than that of the protocol processor; hence, the host spends more time transferring TSDUs than does the protocol processor.

Other pitfalls of protocol processor driving are relatively insignificant in our architecture. From the profiles of chapter five, we can see that the overhead due to the management of shared data structures is only visible in the buffer allocation of large ASDUs. Even in this case, the overhead of 109 µs is insignificant given the fact that the large ASDUs causing such buffer overhead incur a vastly greater cost in their backplane copies. In addi-

tion, some device driver code exists to manage connection state; however, its overhead is also not significant, regardless of ASDU size. Similarly, the effect of multiplexing and demultiplexing command and status information was never observed to be a significant contributor to host processing time. The only ill effects of such multiplexing are the code complexity it introduces into the kernel device driver. The notification of command completion through UNIX signals did incur a visible amount of overhead (143 μs); however, such overhead was not particularly significant, even for API primitives with short ASDUs.

7.3.3. Increased Data Path Complexity

Another pitfall unique to off-host communication architectures is the increase in data path complexity due to addition of the protocol processor device. Our architecture commits these faults, much to the detriment of its user-level throughput and end-to-end latency performance. The concerns here are the use of a low bandwidth connection to the protocol processor, additional data copies, and additional bus contention.

The pathway to our protocol processor is indeed low bandwidth. First of all it is via an I/O backplane. This implies that it is not optimized for high speed as opposed to memory buses on a motherboard. Furthermore, we perform programmed I/O copies of the data, and, because of this, we cannot use the block transfer mode of the VMEbus. As a result, we achieve relatively poor VMEbus throughput for our architecture. Note as well from chapter five that the VMEbus is not alone responsible for the low throughput; factors such as the CPU speed and memory access time also play a role here.

Due to the fact that we buffer our TSDU data on the local memory of the protocol processor board, we incur an extra copy than would otherwise be the case if the protocol processor handled TSDUs resident in host memory. This additional copy causes an extra step in common-case communication processing. Worse yet, its overhead is incurred by the host CPU, thus contributing to host load. Such added overhead has an inescapable effect on user-level end-to-end latency and is quite significant for large ASDUs. If performed in par-

allel with protocol processor data transfer, the ill effects of the extra copies on user-level throughput could be hidden; however, such cannot be the case in our architecture for reasons discussed below.

Since our architecture uses two separate VMEbus modules for the protocol processor and network interface, the architecture suffers bus contention. The detrimental effects of this contention are graphically illustrated in Figure 5.4 of section 5.5.2. One can see that much of the possible parallelism between the host and protocol processor is prevented for large ASDUs due to the fact that both processors wish to perform bus transfers simultaneously. This contention ultimately results from a combination of two factors: (1) the buffering of TSDUs in the protocol processor's local memory, requiring VMEbus transfers between the host and protocol processor, and (2) the placement of the network interface on a separate VME module, requiring VMEbus transfers between and the protocol processor. If either factor were not the case, this contention would be severely decreased, if not eliminated altogether.

These results argue strongly for integrating the network interface on the protocol processor device or eliminating TSDU buffering on the protocol processor. Taken together with observations of the low bandwidth data path and problems of the extra copy, our results argue most strongly for eliminating the extra buffering. Ideally one would eliminate both the buffering and the separation of the protocol processor and network interface in to different bus modules.

7.4. Critique of the Simple Performance Model

The architecture-based performance model developed in chapter six was designed to easily allow both performance evaluation of existing off-host communications architectures and performance predictions for off-host architectures undergoing their design. To provide such services, the model takes a set of input parameters and predicts a set of output quantities. The guiding design principles for the model were simplicity and practicality. For

the model to be judged as simple, its input parameters must be easily obtainable by the model's user. Furthermore, the mathematical analysis of the model must be simple enough for the user to apply. Due to the use of time-average measurement of architectural inputs and the simple mathematical analysis of protocol performance plots, the model's inputs are relatively easy to obtain. The greatest difficulty in parameter gathering is that, to gain some of the architectural parameters, knowledge of UNIX character device drivers is required. However, we argue that such knowledge is mandatory for UNIX communications system designers in any case. What is most important is that the input parameter gathering requires no precise measurement equipment and no probability distributions of input quantities.

Our model is also quite easy to apply. Due to the avoidance of probability distributions for input parameters, we sidestep the complex and often intractable mathematics of queueing theory. Rather, through the use of averages quantities, our mathematics is quite simple, involving only algebraic expressions. Although our analysis is based upon the assumption of a specific class of architecture, the methodology is so general that it is applicable to other classes as well. Several examples were given in section 6.10.1 to illustrate this.

The practicality of the model must be judged based upon both the performance quantities it predicts and how accurately it predicts them. Clearly, user-level end-to-end latency and throughput are quantities of interest to the users of the model. Thus, in predicting these quantities, the model is indeed performing a valuable service. In addition, the prediction of host load is of particular interest to the designers off-host communication architectures since the reduction of host load is one of the unique benefits of such architectures. A possible criticism, however, of our host load prediction is the choice of its definition. One may argue that another metric besides seconds of CPU time is more appropriate for characterizing host load. For example, one may prefer the metric of the percentage of host CPU utilization or a percentage of host cycles out of the overall processing time of the communications architecture. In defense of our load prediction metric, we argue that such

a prediction is quite fundamental and may therefore be used to derive the many other metrics, which others may suggest.

One may also argue that other predictions, such as interrupts per ASDU, should be included as model outputs; however, such quantities are more likely to be intrinsic to the design of an off-host architecture than derived from such a design. Similarly, the bound on incident ASDUs is a quantity inherent in a particular off-host design.

One may also argue that our model should predict the benefits to the transport protocol due to off-host implementation. However, the complexity of transport protocol analysis may likely result in a complex model that is prohibitively difficult for users to apply. If a simple approach could be adopted, perhaps a separate model predicting the benefits of dedicated cycles, custom hardware, and a specialized operating system could be constructed.

The accuracy of the model is acceptable when one appreciates the fact that it requires no specific details of the implementation of an architectural design. Admittedly, the model predictions for short ASDUs are not particularly accurate. At its worst, the model under-predicts the host load due to a one-byte `SEND_MESSAGE` primitive by 36%. The latency and host load predictions for short ASDUs are lower than observed results due to the fact that, as applied, the model only takes into account the inherent overhead of the architectural components and protocol processor. Since we do not consider the additional delay effects introduced by the software that operates within the architectural framework, it is clear that our model predictions should indeed be low.

Our model over-predicts the synchronous throughput short ASDUs. This is due to the fact that the ASDU length-independent overhead is under-predicted. A discussed above, this discrepancy is present because ours are *a priori* predictions. Such estimates do not assume knowledge of software overhead of the implementation of our architecture.

For other reasons, the asynchronous throughput prediction is poor for short ASDUs. This is due to the assumptions on which the formula providing the predictions is based.

Such assumptions assume the presence of bus contention and are claimed to be valid only for sufficiently large ASDUs that result in the saturation of the backplane from large amounts of traffic.

On the other hand, our model predicts rather well for large ASDUs. The graphic comparisons in section 6.8.2 show that synchronous and asynchronous user level throughput and end-to-end latency predictions are much more accurate in the large-ASDU case. Likewise, the accuracy of our host load prediction for a 64-KB `SEND_MESSAGE` primitive is within 7.1%. The reason why such predictions are more accurate is that, for large ASDUs, the overhead of software implementation is dwarfed by the backplane transfer and protocol processing times, quantities which are indeed taken into account in the model.

If the user of the model wishes increased accuracy for short ASDUs he may apply his knowledge of the overhead of the software implementation of the modelled architecture as it becomes known. This approach was in section 6.10.2.

It is not clear that the use of queuing theory would significantly improve the predictions for short ASDUs. It is our belief that these predictions were largely affected by the fact that we ignored implementation-dependent parameters in our analysis, rather than by our use of simple mathematical analysis.

7.5. Suggestions for Future Work

There are many avenues for future work in both the area of off-host communication architectures and the area of the performance modelling of such architectures. It is our opinion that more fruitful work can be done via the redesign of our architecture for SAFENET. The top three areas in which our architecture could be improved are in: (1) the avoidance of buffering ASDUs, (2) the integration of the protocol processor and network interface on the same circuit board, and (3) the avoidance of system calls. To pursue these modifications, some new hardware choices would be necessary. In addition, one would have to either (a) give up the goal of secure access (in the sense of multiprogramming) to the pro-

tocol processor, or (b) find a way to provide security without using system calls in the common case of communication. The latter solution could perhaps be achieved though the use of mutually-private yet protocol-processor-shared memory regions for each process using the protocol processor.

Additional work on the performance model is also possible. There are many extensions presented in section 6.10 which await pursuit. In particular, it would be of great interest to see if the use of implementation-dependent knowledge can significantly improve the model predictions for short ASDUs. With such results in hand, we may decide whether a more complex model is necessary in order to accurately predict performance for short ASDUs.

If queueing theory were employed in a model of off-host communications architectures, it would be possible to characterize the facets of the architecture exhibiting stochastic behavior. The following areas of architecture would benefit most greatly from queueing theory analysis and probabilistic description:

- The delay introduced in the FN and TN queues between the host and protocol processor
- The effect of limits on the length of the FN and TN queues
- The effect of limits on the maximum number of operations outstanding on a connection
- The stochastic behavior of the protocol processor, accounting for the variance in both latency and throughput
- The stochastic behavior of the backplane bus, including the probability and duration of blocking due to contention or the variance in arbitration times
- The effects of bus arbitration and release policy on the stochastic behavior of the bus
- The effects of other processes on the host

We welcome any further attempts to model off-host communications architectures in hopes that they reveal more insight in to the impact of design choices and allow better evaluation and prediction of off host architectures.

7.6. The Suitability of Off-Host Communications Architectures

Taking into account both our research in off-host communications architectures and the work of others, we make a few general comments about the suitability of off-host architectures. Our results suggest that such architectures are useful in two general scenarios: (1) when the host CPU cannot sustain the burden of protocol processing and still meet the needs of its application processes, and (2) when the protocol must provide a higher level of service than can be achieved by the host computer and operating system. There are, however, some caveats for the designers of off-host architectures which, if left unheeded, can erode the benefits of such architectures.

In some situations the host CPU may execute a set of processes which have real-time requirements. In such cases the addition of more load on the CPU or the presence of host processing with a nondeterministic execution time could result in the unacceptable performance of other host tasks. Our architecture shows that off-host architectures avoid such problems and are therefore suitable in these real-time scenarios. Although our off-host architecture occasionally suffers significant host load due to programmed I/O, other architectures (e.g., that of Beach in section 3.6) which avoid such load are possible. Furthermore, we have established that our architecture has both deterministic host execution times and bounded interrupt arrivals, thereby providing bounded host processing times. Thus it is clear that off-host architectures are suitable for meeting several real-time host processing demands.

In many cases, protocols which execute on-host are incapable of meeting the minimum service requirements of applications due to the constraints imposed by the host system hardware, its operating system, or other host processing. Our work indicates that the use of an off-host communications architecture can solve some of these problems. The comparison of our off-host architecture with an in-kernel architecture shows that the host computer's hardware and operating system architecture greatly constrained transport protocol throughput. Our off-host architecture's use of a lightweight operating system allowed

the off-host approach to achieve higher user-level throughput. Such results suggest that off-host architectures are more suitable for modern, high-throughput LANs. In addition, our protocol processor is always prepared to process protocol traffic through the use of its dedicated cycles. As a result, we think that it is better able than the host to handle the real-time and/or continuous-media traffic carried by modern LANs.

Our results also indicate that the designers of an off-host communications architecture must watch out for various pitfalls. In our work we found the most significant of these to be system call overhead and bus transfer time. These results along with the body of related work on off-host architectures suggest that, in the design of such architectures, one should seek to minimize (1) the number of architectural layers in the design, (2) the overhead at each layer, and (3) the overhead of inter-layer communication. Our experience suggests that the overhead of inter-layer communication is the most significant detriment to the suitability of off-host communications architectures. One must also be willing to incur the monetary cost of the additional protocol processing hardware and grapple with the complexity such hardware introduces.

In short, off-host communications architectures can be used to the practical advantage of both a host computer system and its communications protocols. When built for the performance-critical types of systems described above and with the proper architectural design choices, such architectures are beneficial to all host applications.

References

[1] M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian Jr., and M. W. Young, "Mach: A New Kernel Foundation of UNIX Development," *Proceedings of the Summer 1985 USENIX Conference*, July 1986, pp. 93-113.

[2] American National Standards Institute/Institute of Electrical and Electronics Engineers, *VMEbus Specification Manual*, STD 1014-1987, VMEbus International Trade Association, Scottsdale, Arizona, 1987.

[3] B. Beach, "UltraNet: An Architecture for Gigabit Networking," *Proceedings of the 15th Conference on Local Computer Networks*, Minneapolis, Minnesota, September 30-October 3, 1990, pp. 232-248.

[4] A. G. Bell and G. Borriello, "A Single Chip NMOS Ethernet Controller," *Proceedings of the IEEE International Solid-State Circuits Conference*, February 1983, pp. 70-71.

[5] P. L. Borrill, "32-Bit Buses—An Objective Comparison," *Proceedings, BUSCON 1986 West*, San Jose, California, 1986, pp. 138-145.

[6] T. Braun and M. Zitterbart, "A Parallel Implementation of XTP on Transputers," *Proceedings of the 16th Conference on Local Computer Networks*, Minneapolis, Minnesota, October 1991, pp. 321-329.

[7] T. Braun, B. Stiller, and M. Zitterbart, "XTP and VMTP on Multiprocessor Architectures," *Proceedings of the International Workshop on Advanced Communications and Applications for High-Speed Networks*, Munich, Germany, March 1992, pp. 67-76.

[8] D. R. Cheriton, "VMTP: A Versatile Message Transaction Protocol," Technical Report RFC 1045, Defense Advanced Research Projects Agency, February 1988.

[9] G. Chesson, "The Protocol Engine Project," *Unix Review*, September 1987.

[10] G. Chesson, "XTP/PE Design Considerations," H. Rudin and R. Williamson, Ed., *Protocols for High-Speed Networks*, Elsevier Science Publishers B. V., North Holland, 1989, pp. 27-33.

[11] E. C. Cooper, P. A. Steenkiste, R. D. Sansom, and B. D. Zill, "Protocol Implementation on the Nectar Communication Processor," *Proceedings, SIGCOMM '90*, Philadelphia, PA, September 1990, pp. 135-144.

[12] P. Coquet, "GAM-T-103 Reference Model for Military Real-Time Local-Area Networks (MRT-LAN)," Proceedings of the IFIP Workshop on Protocols for High-Speed Networks, Zürich, May 1989.

[13] B. J. Dempsey, J. C. Fenton, J. R. Michel, A. S. Waterman, and A. C. Weaver, "Tutorial on UVA SAFENET Lightweight Communications Architecture," Computer Science Technical Report Number TR-93-01, University of Virginia, January 1993.

[14] B. J. Dempsey, J. C. Fenton, J. R. Michel, A. S. Waterman, and A. C. Weaver, "Ada Binding Reference Manual—SAFENET Lightweight Application Services," Computer Science Technical Report TR-93-02, University of Virginia, January 1993.

[15] B. J. Dempsey, J. C. Fenton, J. R. Michel, A. S. Waterman, and A. C. Weaver, "SAFENET Internals," Computer Science Technical Report Number TR-93-05, University of Virginia, January 1993.

[16] C. Diot and M. N. X. Dang, "A High-Performance Implementation of OSI Transport Protocol Class 4; Evaluation and Perspectives," *Proceedings of the 15th Conference on Local Computer Networks*, Minneapolis, Minnesota, October 1990, pp. 223-230.

[17] C. Diot and V. Roca, "XTP/KRM Implementation on a Transputer Network," *Proceedings of the 16th Conference on Local Computer Networks*, Minneapolis, Minnesota, October 1991, pp. 310-320.

[18] R. Duncan, Ed., "The MS-DOS Encyclopedia," Microsoft Press, Redmond, Washington, 1988.

[19] I. Erickson, "Protocol Controller Chip Manages X.25 Interface," *Computer Design*, Vol. 24, September 1, 1985, pp. 78-81.

[20] D. Giarrizzo, M. Kaiserswerth, T. Wicki, and R. C. Williamson, "High-Speed Parallel Protocol Implementation," H. Rudin and R. Williamson, Ed., *Protocols for High-Speed Networks*, Elsevier Science Publishers B. V., North Holland, 1989, pp.165-180.

[21] D. T. Green and D. T. Marlow, "SAFENET - LAN for Navy Mission Critical Systems," *Proceedings of the 14th Conference on Local Computer Networks*, Minneapolis, Minnesota, October 1989, pp. 340-346.

[22] A. N. Habermann, "Synchronization of Communicating Processes," *Communications of the ACM*, Vol. 15, No. 3, March 1972, pp. 171-176.

[23] B. Heinrichs, "XTP Specification and Parallel Implementation," *Proceedings of the International Workshop on Advanced Communications and Applications for High-Speed Networks*, Munich, Germany, March 1992, pp. 77-84.

[24] International Organization for Standardization, "Information Processing Systems—Open Systems Interconnection—Basic Reference Model," Draft International Standard 7498, October 1984.

[25] N. Jain, M. Schwartz, and T. R. Bashkow, "Transport Protocol Processing at BGPS Rates," *Proceedings, SIGCOMM '90*, ACM, New York, 1990, pp. 188-199.

[26] D. Julin, MACH Networking Group, “Network Server Design,” Technical Report, Carnegie Mellon University, September 1989.

[27] H. Kanakia, D. R. Cheriton, “The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors,” *Proceedings of the SIGCOMM ‘88 Symposium on Communications Architectures and Protocols*, August 1988, pp. 175-187.

[28] A. S. Krishnakumar and K. Sabnani, “VLSI Implementations of Communication Protocols—A Survey,” *IEEE Journal on Selected Areas in Communications*, Vol. 7, No. 7, September 1989, pp. 1082-1090.

[29] L. Lamport, “Synchronization of Independent Processes,” *Acta Informatica*, Vol. 7, No. 1, 1976, pp. 15-34.

[30] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[31] R. A. MacClean and S. E. Barvick, “An Outboard Processor for High Performance Implementation of Transport Layer Protocols,” *Proceedings, GLOBECOM ‘91*, 1991, pp. 1728-1732.

[32] MIL-STD-2204: *Survivable Adaptable Fiber Optic Embedded Network (SAFENET)*, United States Department of Defense, September, 1992.

[33] R. J. Mitchell, E. T. Saulnier, and M. C. Orlovsky, “A Partitioned Implementation of the Xpress Transfer Protocol (Part I),” *Proceedings of the 16th Conference on Local Computer Networks*, Minneapolis, Minnesota, October 1991, pp. 301-309.

[34] R. J. Mitchell, E. T. Saulnier, “Experience with an XTP Implementation for Embedded Systems,” *Proceedings of the 17th Conference on Local Computer Networks*, Minneapolis, Minnesota, September 1992, pp. 586-592.

[35] A. N. Netravali, W. D. Roome, K. Sabnani, “Design and Implementation of a High-Speed Transport Protocol,” *IEEE Transactions on Communications*, Vol. 38, No. 11, November 1990, pp. 2010-2024.

[36] J. L. Paige, “SAFENET - A Navy Approach to Computer Networking,” *Proceedings of the 15th Conference on Local Computer Networks*, Minneapolis, Minnesota, September 30-October 3, 1990, pp. 268-273.

[37] D. M. Ritchie, “A Stream Input-Output System,” *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, October 1984, pp. 1897-1910.

[38] E. T. Saulnier and R. J. Mitchell, “A Multi-Processor Partitioning of XTP,” *Proceedings of the International Workshop on Advanced Communications and Applications for High-Speed Networks*, Munich, Germany, March 1992, pp. 85-92.

[39] M. Siegel, M. Williams, and G. Rößler, "Overcoming Bottlenecks in High-Speed Transport Systems," *Proceedings of the 16th Conference on Local Computer Networks*, Minneapolis, Minnesota, October 1991, pp. 399-407.

[40] A. Silberschatz, "Communications and Synchronization in Distributed Programs," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 6, November 1979, pp. 542-546.

[41] R. Simoncic, A. C. Weaver, and M. A. Colvin, "Experience with the Xpress Transfer Protocol," *Proceedings of the 15th Conference on Local Computer Networks*, Minneapolis, Minnesota, September 30-October 3, 1990, pp. 123-131.

[42] M. Stark, A. Kornhauser, and D. Van-Mierop, "A High Functionality VLSI LAN Controller for CSMA/CD Networks," *Proceedings of the IEEE Compcon Spring*, February 28-March 3, 1983, pp. 510-517.

[43] P. Steenkiste, "Analyzing Communication Latency Using the Nectar Communication Processor," *Proceedings of SIGCOMM '92*, 1992, pp. 199-209.

[44] W. T. Strayer, B. J. Dempsey, and A. C. Weaver, *XTP: The Xpress Transfer Protocol*, Addison-Wesley, Reading, Massachusetts, 1992.

[45] Sun Microsystems, Inc. "Sun 4300 CPU Board Hardware Reference Manual," Part No: 800-3081-05, Mountain View, California, April 1989.

[46] Sun Microsystems, Inc., "Writing Device Drivers," Part No: 800-3851-10, Mountain View, California, March 1990.

[47] A. S. Waterman, "A Comparison of Off-Host vs. In-Kernel Communications Architecture," M.S. Thesis, Department of Computer Science, University of Virginia (in preparation).

[48] 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, No. 2, May 1987, pp. 97-120.

[49] C. M. Woodside and R. G. Franks, "Alternative Software Architectures for Parallel Protocol Execution with Synchronous IPC," IEEE/ACM Transactions on Networking, Vol. 1, No. 2, April 1993, pp. 178-186.

[50] M. Zitterbart, "A Multiprocessor Architecture for High-Speed Network Interconnections," Proceedings, IEEE INFOCOM '89, 1989, pp. 212-218.

[51] M. Zitterbart, "High-Speed Protocol Implementations Based on a Multiprocessor Architecture," H. Rudin and R. Williamson, Ed., *Protocols for High-Speed Networks*, Elsevier Science Publishers B. V., North Holland, 1989, pp. 151-163.

[52] M. Zitterbart, “High-Speed Transport Components,” *IEEE Network Magazine*, Vol. 5, No. 1, January 1991, pp. 54-63.

Appendix A: Time-Average Measurement of Architectural Input Parameters

A.1. Rationale for Time-Average Measurement

This appendix discusses a simple method for obtaining the values of architectural input parameters through *time-average measurement*. We first explain what is meant by a time-average measurement. A time-average quantity is one which results from a measurement of the total time to perform several samples of identical operations. The time-average quantity is simply a statistic measuring the ratio of this total time to the number of samples—it represents the average time to perform a single operation.

In the simple performance model presented in chapter six, we make use of several input parameters (δ_w, δ_r, δ_i, ρ_w, and ρ_r) to characterize the performance of the components of our specific class of off-host architecture. In order to make our performance model easy to apply, we must allow its users to obtain the small but descriptive set of architectural parameters in a simple manner. We feel that the use of the technique of time-average measurement is best for this purpose.

The reasons for using time-average measurement are due to its simplicity and practicality relative to the alternative method of direct measurement. Direct measurement of the execution time of an operation is an inadequate choice for our model for two reasons. First, the granularity of the measurement devices present in most computer laboratory environments is on the order of tens of milliseconds; however, the quantities we wish to measure are on the order of microseconds and even nanoseconds. Thus it is typically impossible to measure such quantities without obtaining more powerful (and often expensive) timing devices.

The second reason is as follows. Even if precise measurements were possible, they would either indicate: (1) a deterministic time value for the measured operation or (2) a distribution of several time values. In the first case, time-average measurement would yield

the same value as the directly-measured value; hence, the direct measurement would provide no practical advantage other than imparting the knowledge that the time of the operation is deterministic. In the second case, the analysis of a model using probability distributions would require advanced mathematics (e.g., queuing theory). Such difficult analysis is contrary to our stated goal of simplicity.

A.2. Test Harness

To perform time-average measurement of the model parameters in chapter six, we used the test harness listed in Figure A.1. The test harness is simply an example of a C program which measures the total time required to perform several samples of an operation, and reports the measured total time and the calculated time-average for a single operation. The operation under consideration in our test harness is represented by the symbol *operation()*. In an actual test, we would substitute the particular operation that we wish to measure for this symbol. The `gettimeofday()` system call employed in the code allows it to read the current time in seconds (`tv_sec`) and microseconds (`tv_usec`). The measurement of specific model parameters is discussed in the next section.

```
#include <sys/time.h>
#include <stdio.h>

#define SAMPLES 1000000

void main()
{
    struct timeval   start, finish;
    double           total_sec;
    int              i;

    gettimeofday(&start, NULL);
    for (i = 0; i < SAMPLES; i++) operation();
    gettimeofday(&finish, NULL);

    total_sec = ((double)finish.tv_sec + (double)finish.tv_usec/1.0e6) -
        ((double)start.tv_sec + (double)start.tv_usec/1.0e6);
    printf("total sampling time = %f sec\n", total_sec);
    printf("average sample time = %f sec\n", total_sec/(double)SAMPLES);
}
```

Figure A.1: Test Harness

In order to acquire a time-average measurement with a desired number of significant digits, the number of samples performed by the test harness must be chosen properly. This number is represented in our test harness by the symbol `SAMPLES`. If we desire that our time-average measurement have *n* valid digits, the number of samples must be chosen such that the measured total sampling time contains at least *n* valid digits. Such digits result from updates of the system clock through clock ticks; thus we must ensure a minimum number of clock ticks to reach a given degree of accuracy. For example, our system updates one digit of the clock on every tick; thus, if we desire three digits of precision we must ensure that our test runs for at least 10^3 clock ticks. Since our clock granularity is 10 ms (10^{-2} seconds/tick) we must make certain that the total amount of time taken to perform our time-average measurement is at least $(10^3 \text{ ticks})\,(10^{-2} \text{ seconds/tick}) = 10 \text{ seconds}$.

A.3. Measurement of Specific Parameters

In this section we describe how to use the time average measurement technique discussed above to calculate the specific architectural parameters presented in chapter six. The measurement of such parameters requires some knowledge of the organization and configuration of UNIX character device drivers. It is not unreasonable to expect the users of our model to have such knowledge since such sophistication is mandatory for the developers of an off-host communications architecture in a UNIX environment. The device driver we refer to is the one which shall be eventually be used to communicate with the protocol processor. It needn't be completely implemented; however, it must be configured into the operating system kernel.

We also refer below to the l_c and l_a parameters. These parameters are not measured; rather, they are implied by the choice of data structures for the control and acknowledgment blocks. When using the C programming language for implementation, the values of l_c and l_a may be determined by applying the `sizeof` operator to the types of these data structures.

A.3.1. System Call Overhead

In order to calculate the round-trip times for the `write()` and `read()` system calls, δ_w and δ_r, we start by simply using such operations in our test harness. The `write()` must be to the protocol processor device driver, and it must pass an l_c-byte buffer. Similarly the `read()` must pass an l_a-byte buffer to the protocol processor device driver. In addition, some initial work must be performed so that the system calls reach the character device driver routines used for their implementation and then immediately return from the call. Such work is quite straightforward and is far short of what is required to implement an entire architectural design. To measure δ_w, we place an abrupt return from the device driver at the first line of the routine which is to implement the `write()` operation. In our implementation, this is the `strategy()` routine, which is called by `physio()`. Since we also implement the `read()` operation in `strategy()`, we use this same abrupt return to measure δ_r as well.

A.3.2. User-Level Interrupt Overhead

It is relatively complicated to measure the round-trip time of a UNIX signal. This is because we must have the device driver, which resides in the kernel, send the signal to the user process performing the measurements. To do this we chose to use a `write()` system call as the operation in the test harness. This call is used simply to initiate the device driver so that it may send signals to the process. At the first line of the device driver's implementation of `write()`, we use the `psignal()` kernel library function to send a `SIGIO` signal to the calling process (i.e., the test harness). The test harness features a simple signal handling routine which returns immediately. This test harness has the effect of measuring the round-trip signal handing time and the round-trip time for a `write()` system call; thus, to arrive at δ_i, the round-trip time for the signal, we need only subtract δ_w.

A.3.3. Bus Transfer Overhead

To measure the ρ_w and ρ_r parameters, those which characterize the cost of bus transfers, we placed our entire test harness in the kernel device driver. In this context we could not issue any system calls; hence, we could not use the `gettimeofday()` system call to read the time. Instead we used the kernel's `time` variable, a global data structure that stores the current time of day in both seconds and microseconds. In order to enter kernel mode, we have a user process issue a `write()` of *n* bytes to the device driver, which contains the test harness code in its `write()` routine. To measure ρ_w, we use the `copyin()` kernel utility function as the operation in our test harness. The `copyin()` function copies the *n*-byte user-level buffer specified in the system call to the shared memory on the protocol processor board. The test harness gives us the average time in seconds to perform the *n*-byte copy; hence, we obtain ρ_w by dividing the total time by *n*. The measurement of ρ_r proceeds in a similar manner except that we use the `copyout()` function as our measured operation; `copyout()` copies from the on-board protocol processor memory to the *n*-byte user-specified buffer provided in the `write()` system call.

A.4. Elimination of "Noise"

When performing the measurements described above, one must be careful to avoid interference from other factors which could confound the measurements. These factors include the execution of other processes or hardware interrupts. To avoid such problems we recommend that the time average measurements be performed with an unloaded system. In addition, any periodic system daemons which are not critical to short-term system operation (such as `update` on SunOS) should be eliminated. Furthermore, on systems such as ours where the clock is updated via tick timer interrupts, we one must ensure that such interrupts are never disabled at any point during the measurements.

Appendix B: Extrapolation of Protocol Processor Input Parameters

B.1. Rationale for Protocol Processor Performance Plots

This appendix discusses a simple method for obtaining the values of the protocol processor input parameters presented in chapter six through the use of protocol processor performance plots. The types of plots we are concerned with are those that both occur commonly in performance studies and serve to illustrate the end-to-end latency or throughput of the protocol processor versus TSDU size. The end-to-end latency graph is used directly; in contrast, the protocol processor throughput graph is used indirectly to derive a protocol processor period graph.We choose to use only the latency and period plots because such plots exhibit a nearly linear shape. We may then fit linear equations to these plots and use the coefficients of the equations to characterize protocol processor performance.

The main advantage of this approach is that protocol performance plots (1) are often provided with protocol implementations and (2) are otherwise derivable from other descriptions of protocol performance (e.g., tables). We chose to fit lines to these plots rather than more complicated curves because the curves themselves are quite linear in the first place. The resulting protocol parameters represent the aspects of protocol processor performance that are most useful for our simple model.

B.2. Protocol Processor Input Parameter Calculation

In this section we calculate the protocol processor input parameters. These parameters are quite simple to calculate; one need only fit lines to the given and derived protocol processor performance plots and then find the equations of these lines. The protocol parameters we seek are simply the slopes and vertical-axis intercepts of the equations.

In order to calculate the protocol processor delay parameters, δ_0 and ρ_δ, we obtain a plot of the end-to-end latency of the protocol processor, represented here by δ, versus

TSDU length, represented here by *l*. Figure B.1 contains an example of such a plot. Given such a plot, we simply fit a line to it as shown in the figure. The coefficients of the equation of the line, $\delta = \rho_\delta l + \delta_0$, are simply our desired delay parameters.

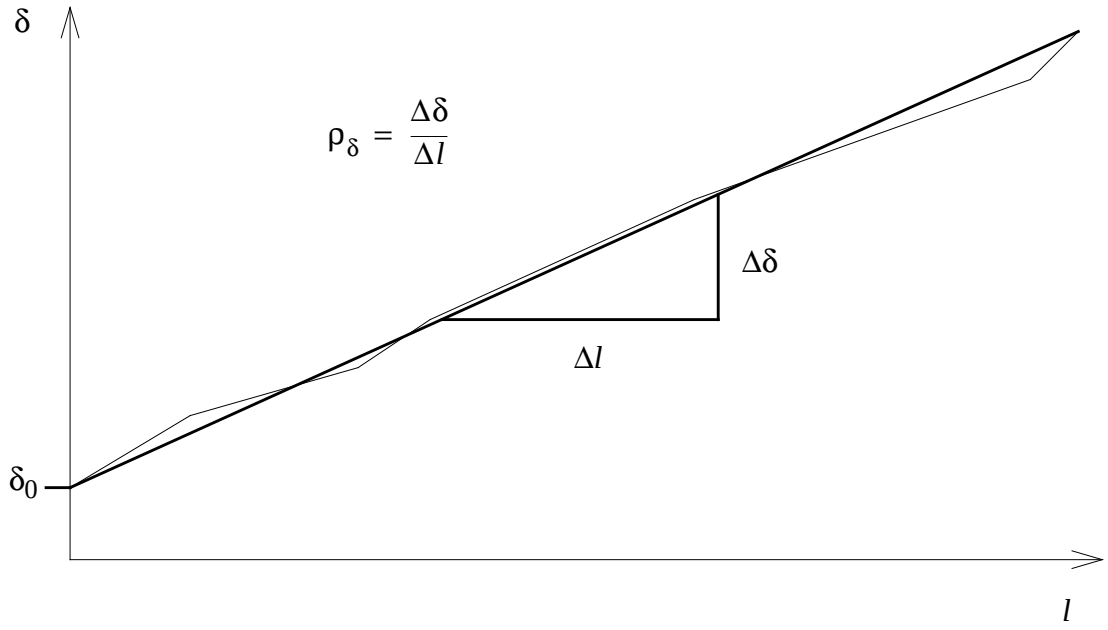


Figure B.1: Fitting a Line to a Plot of Protocol Processor Latency vs. TSDU Size

To calculate the protocol processor period parameters, T_0 and ρ_T, we must first derive a plot of the protocol processor period versus TSDU length from a given plot of protocol processor throughput versus TSDU length. To do this we start by taking a subset *S* of points (l_i, τ_i) from the protocol processor throughput graph. Here l_i is the TSDU length for the *i*th such point and τ_i is the throughput corresponding to l_i. From *S* we calculate the set *S*' of points (l_i, T_i) which will make up the protocol processor period graph. Here, T_i represents the protocol processor period corresponding to l_i. We then derive each point (l_i, T_i) in *S*' from a corresponding point (l_i, τ_i) in the set *S* using the relation $T_i = l_i / \tau_i$. Now the points of *S*' may be plotted in a graph illustrating protocol processor period, represented here by *T*, versus TSDU size, represented by *l*. The equation of the line fitting this curve has the form $T = \rho_T l + T_0$, yielding the period parameters ρ_T and T_0.