A LANGUAGE-BASED APPROACH
TO PROTOCOL IMPLEMENTATION

(Ph.D. Dissertation)

Mark Bert Abbott

TR 93-24

August 10, 1993

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

This research was sponsored in part by DARPA Contract DABT63-91-C-0030.

A LANGUAGE-BASED APPROACH TO PROTOCOL IMPLEMENTATION

by

Mark Bert Abbott

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 3

ACKNOWLEDGMENTS

I gratefully acknowledge my advisor, Larry Peterson, for his creative ideas, uncanny sense of research significance, relaxed attitude, and confidence in my abilities.

I thank the faculty, especially Greg Andrews, Mary Bailey, Saumya Debray, Norm Hutchinson, and Rick Schlicting, for their guidance, and their ability to balance technical excellence with concern for people. I also thank our outstanding lab and office staffs for a level of support that I am unlikely to see equaled. I thank Wendy Swartz for her friendship as well as for helping me navigate bureaucratic hurdles.

I thank all the members of the Network Subsystems Research Group for providing a fertile and supportive research community. I am particularly indebted to Peter Druschel for both the benefit of his exceptional technical abilities and his moral support.

I thank my fellow graduate students, Larry Brakmo, Curtis Dyreson, Clint Jeffery, Nick Kline, Jim Knight, Dave Lowenthal, Shamim Mohamed, Michael Pagels, Mike Soo, and Vic Thomas, for their friendship. I am especially grateful to Tyson Henry and Mudita Jain for all they have done for me. And, like every U. of A. Computer Science graduate student, I owe a debt of gratitude to Patrick Homer, unofficial mayor of CS graduate students, for all he does to make that community cohesive and enjoyable.

I thank John Peterson for first putting Arizona on my map, and later sharing climbing adventures.

Finally, I thank my parents for their constant love and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This thesis explores two strategies for supporting the development of network communication software: imposing constraints on protocol design at the specification level, and using a special-purpose language for protocol implementation. It presents a protocol implementation language called Morpheus. Morpheus utilizes the new strategies to provide a higher level of abstraction, finer grain modularity, and greater software reusability than previous approaches.

Morpheus is able to provide a high level of abstraction because of built-in knowledge about its problem domain. It has a narrow problem domain—network protocols—that is further narrowed by the application of specification-level constraints. One particular constraint—the *shapes* constraint, which partitions protocols into three basic kinds—is particularly effective in raising the level of abstraction.

Morpheus's support for modularity and, indirectly, software reuse hinges on reducing the performance penalty for layering. When protocol layering entails a high performance cost, developers are motivated to build complex monolithic implementations that are hard to design, implement, debug, modify, and maintain. Morpheus reduces the performance costs of layering by applying optimizations based on common patterns of protocol execution. If the degree of modularity is held fixed, then the optimizations simply improve performance. An optimization based on Integrated Layer Processing is particularly noteworthy for its dramatic contribution to network throughput while preserving modularity.

CHAPTER 1
INTRODUCTION

Computer networks are systems of interconnected computers. Interconnecting computers makes it possible to share resources such as data, programs, and specialized hardware. Communicating data between programs and between people has grown to rival computation as the primary function of computing systems. Computer networks also offer increased reliability, through redundant hardware and replicated data and programs, and price advantages relative to large mainframe computers of comparable computing power.

The computers in a network are connected by software as well as hardware. Just as an operating systems provides a virtual machine built on top of a raw physical machine, network software builds sophisticated communication services on top of the primitive communication provided by network hardware. This software is quite complex because of its explicitly distributed nature with the potential for partial failures, because of the heterogeneity of the hardware technologies used to interconnect computers (even within a single network), and because of the variety of distributed applications that must be efficiently supported. Network software is also frequently revised as a result of changing hardware technology, new applications with new communication service requirements, the integration of communications services, and the exponential growth in the number of computers that are networked together.

This dissertation introduces a new approach that supports the development of this complex, performance-critical, frequently revised software.

1.1 Introduction to Network Software

Network software is responsible for data communication and synchronization between processors connected by hardware links. In the context of networking, such processors are referred to as *hosts*. A hardware network typically provides a primitive communication service that is subject to data corruption or loss, and connects only a modest number of hosts. Network software builds more sophisticated communication services on top of the hardware, with better failure characteristics and extended connectivity.

Communication services are most often one-to-one or *unicast* services; communication is from one entity to another. There are also *multicast* or *group communication* services in which communication from one member of a group goes to all the other members of the group.

Communication services sometimes implement a *Remote Procedure Call (RPC)* or *Request-Reply* service in which the initiator of a peer-to-peer communication is blocked until it receives a reply message, thereby providing a procedure call model for communication. More often, communication services provide simple *message passing* services in which each one-way communication is independent.

Network software is generally structured as a hierarchy of layers. Each layer builds a more sophisticated communication service on top of the communication service provided by the lower layers. Layering is a technique for managing the complexity of network software, and also exposes intermediate communication services for direct use.

Each layer represents a network *protocol*. A protocol is a convention for the exchange of *messages*. Abstractly, a message is a finite series of bits. Messages can contain data which is relayed on behalf of higher protocol layers or applications, as well as control information meaningful to the current layer.

The implementation of a protocol on a host, called a protocol *entity*, follows the protocol's message exchange convention to exchange messages with entities of the same protocol on other hosts, called *peers*. Collectively, the entities of a given protocol implement a protocol layer. One-way communication is implemented using an *asymmetric* protocol, in which there are two kinds of entities, sending entities and receiving entities. Two-way communication is implemented using a *symmetric* protocol, in which all the entities implement the same functionality. Symmetric entities are by far the more common.

Each layer transmits its messages via a lower level communication service. The composition of a protocol layer on top of a communication service results in a new communication service. Underneath the lowest protocol layer is a network hardware which provides the lowest level communication service.

The hierarchy of protocol layers need not be linear. A given protocol layer may support multiple higher-level protocols, each of which provides a different communication service. Furthermore, a given protocol layer may use several lower-level communication services, such as when the lower-level services correspond to different local area networks. Hence the protocols on a host form a *protocol graph* as depicted in Figure 1.1.

Protocols are defined by specifications. A specification prescribes the format or *syntax* of a protocol's messages. Messages generally include control information that is interpreted by the receiving peer, and often include data from higher level protocols or applications. The message syntax determines the layout of these elements. Control information is generally affixed at the beginning of higher level data and called a *header*; occasionally it is affixed at the end and called a *trailer*. The specification also prescribes the behavior of an entity in response to events such as the reception of a given type of message, or a request from a higher level layer to transmit a message. Protocol specifications are critical for *interoperability*—correct interaction between peers—because peers may be implemented on different host machine architectures, in the context of different operating systems, in different languages, by different organizations, and by different programmers.

The two primary metrics of network performance are latency and throughput. These have somewhat different meanings depending whether they are applied to the hardware level, or the protocol level.

At the hardware level, a message is a series of uninterpreted bits. Latency is the time

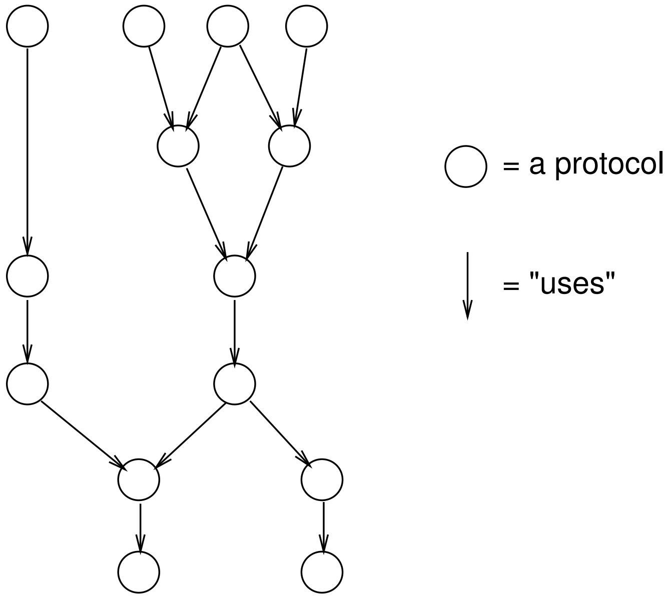


Figure 1.1: A Protocol Graph

elapsed from the sending of a message until the arrival of the first bit of the message at its destination. Intuitively, latency represents a notion of distance, the length of the "pipe" between the source and the destination. Throughput (or, at this level, bandwidth) is the rate at which the bits of a message arrive once the first bit has arrived. If latency is the length of a communication pipe, then throughput is the bore or caliber of the pipe.

Still at the hardware level, the time to transfer a complete message involves both latency and throughput: how long does it take for the first bit to arrive (latency), plus how long does it take for the remaining bits to arrive (the length of the message divided by the throughput). Hence either latency or throughput may dominate depending on the size of the message, latency dominating for short messages, and throughput dominating for long messages.

Latency and throughput have different though analogous meanings at the protocol level. Unlike hardware, protocols distinguish between the different bits of a message, taking different amounts of time to process each of its different parts. The time taken by a protocol to process a message may be modeled as consisting of two components: time to process the header, which may be treated as constant; and time to process the data, which may be treated as linear in the amount of data (the factor is zero for protocols that do not process the data).

At the protocol level, latency is the time elapsed from the sending of a message consisting solely of headers until the arrival of the message at its destination; it captures the component of message transfer time that is independent of message length. Throughput is the average rate at which the bits of long messages arrive; it captures the message length dependent component of message transfer time. In this dissertation, latency and throughput are used in their protocol level senses.

More specialized computer network concepts will be introduced in Chapter 2.

1.2 Network Software is Evolving

Network software is changing in response to new network hardware, new application requirements, the integration of previously disjoint communication systems, and the changing scale of networks.

- Changing network hardware changes the communication services at the foundation of a network architecture. The new hardware may have different failure characteristics, different performance, or a different addressing scheme. It may or may not support different communication models much as multicast. Changing the hardware characteristics generally changes the implementation tradeoffs in higher network software, even in cases where it is possible to nominally provide the same communication services while confining software changes to the software that directly interfaces with the network hardware. For example, a reduction in the rate of bit errors in messages may make it more efficient to check for such errors only in the host for which a message is ultimately destined, rather than checking at each intermediate host that relays the message. Hence, changing the hardware can directly motivate changing the software.

- New applications such as multimedia motivate changes in network software by introducing new communication service requirements. Again network software tradeoffs would change even in cases where it might be possible to confine the software changes outside the network software by building new communication functionality into the application. Hence, network software must adapt to provide the appropriate services.

- Voice and data communication, which were previously provided by disjoint networks, are in the process of being integrated. Network software must change to reconcile these two dissimilar styles of networking.

- The number of networks in existence continues to grow. Many small scale homogeneous networks are interconnected to form *internetworks* which are themselves networks, but large scale and heterogeneous. The foremost internetwork is a global internetwork known as the *Internet* [Com88]. The Internet is experiencing exponential growth that will require network software changes in addressing and routing algorithms.

Thus, developing and modifying network software is an ongoing process. For this reason, and because of the complex, performance-critical nature of the software, there is a potentially large payoff for investing in tools to support protocol development. The goal of this research is to make it easier to develop high performance network software.

1.3 Existing Support for Network Software Development

Network software is usually developed with little or no networking-specific program development support, but this is changing. The two forms of protocol development support that have been explored are protocol frameworks and formal techniques.

1.3.1 Protocol Frameworks

One form of support for protocol development is the *protocol framework*. According to [GNI92], a protocol framework

> defines an implementation and execution environment for communication protocols. There are two parts to the service provided by the framework. The first part is a set of structural guidelines which determine protocol implementation details. [...] Common examples of structural guidelines include the format of communication between protocol modules or layers, and the structure of the protocol state machines. The second part of any protocol framework service is a set of library routines to perform common protocol functions.

System V Streams [TT87] was perhaps the first protocol framework, although it lacks the library routines for common protocol functions. Streams was originally designed to support character I/O, and later extended to support protocols. All protocols provide the same interface to adjacent protocols. This interface is block-oriented: all parameters of an operation, including the identity of the operation itself, are buffered in a block that is passed to the protocol module. The parameters to some operations can include user data, in which case a block corresponds to a message. Each protocol module includes two queues for outstanding blocks, one queue for blocks from higher protocols, and one for blocks from lower protocols. Normally, blocks are queued before being processed by a protocol, but a given protocol may process blocks without any queueing. The arrangement of adjacent protocols is established, and can be modified, at runtime.

The x-kernel [HP91] is a full-fledged protocol framework. It began life as an operating system, but is now a networking subsystem that can be installed in other operating systems. As opposed to Streams's block-oriented protocol interface, the x-kernel's uniform protocol interface is call-oriented: an operation is invoked by calling the corresponding function with appropriate parameters. Operations involving message transfer take a message as one of the parameters. Messages are represented by an abstract data type whose operations are library routines or macros provided as part of the x-kernel. Also provided are countdown timers, which are used to determine whether a message has been lost, and hash tables, which are used to demultiplex message streams.

Avoca [OMa90] is a variant of the x-kernel. The most significant difference is a novel multiplexing scheme to which all Avoca protocols must adhere, which is discussed in Chapter 2.

The Parallel Protocol Framework [GNI92] emphasizes support for parallel protocol execution. In addition to a framework like that of the x-kernel, it provides routines for mutual exclusion management of critical sections, ordering mechanisms for protocols that expect implicit event ordering, and sequence number generation routines to support the ordering mechanisms.

The advantages of a protocol framework are:

Code reuse. This takes two forms. The first is reuse of the support routines, which are used by many or all protocols in the framework. The second is reuse of a given protocol implementation, since a uniform protocol interface allows it to be flexibly composed with different adjacent protocols in different contexts.

Consistency. The consistent structure imposed on protocols makes it easier to develop new protocols and maintain or modify existing protocols.

Performance. Performance of protocols in the framework is promoted by a protocol structure designed for efficiency and use of carefully designed and tuned support routines.

Protocol portability. If all protocol access to operating system functions is mediated by the framework, then all the protocols are portable to any system where the framework has been installed.

1.3.2 Formal Techniques

Formal techniques also offer some support for protocol development. However, these mainly focus on aspects of protocol specification. Specifications are expressed using Formal Description Techniques (FDTs) such as Estelle, LOTOS, or SDL [vB87]. Formal specification of protocols is desirable because it makes requirements precise and unambiguous for everyone involved in the design and implementation process, including automated tools. It also makes it possible to manipulate, analyze, and predict the behavior of the system during the design stage and prior to implementation. FDTs fall into two general categories, state models and sequence models [Pia83]. In state model FDTs, the input/output behavior of a system is defined indirectly by specifying a state machine with input and output. In sequence model FDTs, the input/output behavior of a system is defined directly. Most FDTs cannot fully specify a protocol, so usually some of the specification is informal, and sometimes multiple FDTs are used. The form of the specification determines what formal techniques are applicable in subsequent phases.

A protocol specification may be checked for two kinds of correctness properties. *General* properties are properties that are desirable for every protocol, such as being deadlock-free and not having unexecutable code. *Specific* properties are properties that are related to the functional objective of a particular protocol, such as delivering messages in order. There are a variety of techniques (Protocol Verification Techniques) for checking correctness of a specification. Some are fully or partially automated. In general, a given technique is only applicable to certain FDTs, and can only be used to check certain kinds of properties [Saj85].

There are techniques for estimating the performance of a protocol based on its specification. These are based on simulation models generated from the specification, and queuing theory analysis driven by the specification [Rud85].

Compared with protocol specification and verification, there has been less investigation of formal techniques to support protocol implementation [Cho85]. However, some FDTs, particularly state model FDTs, give strong guidance to implementation. Certain FDTs have associated automatic synthesizers that can output part of a protocol implementation based on the specification. These are relatively low level, implementation-oriented FDTs, and the generated implementation takes the form of a skeleton which must be completed with programmer code.

RTAG [And85] represents a different sort of formal technique. In RTAG, protocols are specified using an attributed grammar. The grammar is directly executable via an RTAG parser, resulting in the appropriate behavior for the protocol. Again, some explicit code is needed. RTAG's performance is not competitive with conventional implementations.

1.4 New Strategies for Supporting Protocol Development

This dissertation proposes two new strategies for supporting protocol development: constraining protocol specifications, and using a special-purpose programming language.

1.4.1 Simplifying Protocol Development by Imposing Constraints

I view protocol frameworks as deriving their power from constraints. They constrain the structure and interfaces of protocols, and in effect constrain many of the low-level implementation details by providing support routines. They preempt a class of implementation decisions from the implementer—those decisions that can be based on knowledge of protocols in general, and do not depend on the particular protocol. Constraining protocols to advantage is possible because of the underlying regularity in the problem domain. Theoretically, a protocol could have an arbitrary structure, use arbitrary interfaces, and apply arbitrary algorithms; but in practice, and with experience, patterns and consensus have emerged regarding good solutions that hold across protocols.

Stated another way, there are two motivations for the constraints imposed by protocol frameworks. First, they are intended to enforce a good design discipline. It has been argued that the development of a new engineering discipline often happens in two phases [Hol91]. In the first phase, the capabilities of tools are expanded to cope with the growing set of problems. In the second phase, tools impose a carefully selected set of constraints on the engineer in order to enforce a design discipline based on accumulated experience. Protocol frameworks reflect their designers' ideas of a good design discipline for protocols.

The second motivation for protocol framework constraints is that it makes possible a more powerful tool. In effect, the more the user is constrained, the more the framework knows about what the user wants to do, and can help the user do it with support routines, for example.

A framework could derive more power by imposing more fundamental constraints, constraints that apply to protocol *specifications*. Doing so would further narrow the design space, thereby simplifying the problem domain. One example of such a constraint would be a constraint on message header formats. Header formats are not implementation decisions; they must be given in a specification because different implementations of a given protocol must agree on the header format in order to interoperate.

Specification-level constraints preempt design choices not only from the final implementors of protocols, but also from designers and standards committees. Thus, a constraint imposed at the specification level might exclude some existing protocols, even standardized protocols. Note however that *excluding a particular specification need not mean excluding the equivalent functionality*—constraints should allow the equivalent functionality to be realized in the form of other protocols or collections of protocols.

Existing protocol frameworks restrict their constraints to implementation internals, thereby limiting their power, in order to support existing and conventional protocol specifications. Avoca [OMa90] is an exception; all Avoca protocols must adhere to a novel multiplexing scheme that impacts protocol specifications. Avoca does not, however, use specification-level constraints as a general strategy.

Imposing constraints on protocol specifications is one of the two high-level strategies explored in this dissertation.

1.4.2 Language Support for Protocol Development

The second new strategy explored in this dissertation is the use of a special-purpose language for implementing protocols. This research focuses not on language design and implementation, but rather on protocol abstractions and protocol-oriented compiler optimizations.

This strategy essentially embeds a protocol framework in a language. Protocol frameworks support protocol development through support for code consistency, performance, code reuse, and portability. A protocol implementation language can potentially extend and add to those advantages in the following ways:

A high level of abstraction. A language can present a seamless, high-level model appropriate for thinking about and concisely expressing protocols. In a protocol framework, much of the implementation detail is exposed and must be specified by the programmer.

Protocol-oriented compiler optimizations. A compiler can perform optimizations based on specific common behaviors of protocols.

Fine-granularity integrated support routines. The support routines that are implemented as functions and macros in protocol frameworks can be an integral part of a language. They can be implemented as language primitives, or in some cases, not visible at the source code level but instead automatically applied where needed. They can have a granularity as fine as an assembler instruction.

Constraint enforcement. A language is the perfect medium for enforcing constraints: satisfying the constraints is equivalent to being a legal protocol program, and any implementation choices below the source language level are in the hands of the compiler. In contrast, the user of a protocol framework can circumvent its implicit constraints, for example by using his or her own algorithm in the place of a support routine.

Portability. Portability of protocols implemented in a protocol framework depend on the programmer adhering to the discipline of allowing all system support to be mediated by the framework. A compiler can ensure that protocols never directly make system calls, they just use the language's runtime system. A compiler and runtime system is provided for each system.

Language-level debugging. Protocol-oriented debugging support can be incorporated in the compiler.

Transparent multiprocessing. A compiler for a protocol implementation language might be able to generate the appropriate locking transparently, so that protocol source code is independent of the degree and style of multiprocessing.

Protection. The boundaries between protocol modules can be enforced using techniques such as static analysis and runtime type checking. This would afford greater flexibility to the mapping of protocols onto address spaces, since arbitrary protocols could be co-located in privileged address spaces. This could in turn lead to higher performance by reducing the frequency with which messages must cross address space boundaries.

1.5 Morpheus

This dissertation presents Morpheus, a model for protocol implementation that is intended to be realized as a programming language. My thesis is that the combination of the two novel strategies employed by Morpheus—constraining protocol specifications, and using a special-purpose language—provides powerful program development support for network software. As evidence, this dissertation will show how these strategies allow Morpheus to support three well-established principles of software development: abstraction, modularity, and software reuse.

1.5.1 Abstraction

Morpheus provides high level abstractions for protocols. A high level of abstraction makes it easier to develop protocols in the following ways.

- There is a seamless model for thinking about protocols. The fundamental network abstractions such as messages and connections are an integral part of the language. In protocol frameworks, many low level implementation details are visible, making it difficult to design at a high level.

- The programmer has fewer implementation details to specify. Morpheus hides the implementations of the abstractions. In a protocol framework, the programmer must make more low level implementation decisions.

- The protocol abstractions embody a design discipline. The programmer is protected from hanging himself or herself with bad implementation choices because those choices are preempted by Morpheus. Protocol frameworks are more limited in terms of the implementation choices they can preempt, and the programmer has more latitude to reject the provided implementations and use his or her own.

- The programs implementing protocols are concise in the sense that protocols are expressed with fewer statements and declarations. This notational economy makes protocol programs easier to understand, write, debug, and modify. Unlike a densely coded APL program, which is hard to understand, Morpheus reduces verbiage by hiding implementation details, which makes a program easier to understand. In a protocol framework much implementation detail is an explicit, visible part of a protocol program.

- The decomposition of functionality into simpler protocols is encouraged by the elimination of redundant programming at each layer. Any protocol behavior that can predictably associated with every protocol is provided by the protocol abstractions instead of being specified by the programmer.

1.5.2 Modularity

A software system is *modular* if it is structured as a collection of parts, called *modules*, that interact only through well-defined interfaces. The advantages of modularity derive from the high degree of independence of the modules. Individual modules can be designed, implemented, and modified independently of each other, possibly in parallel by different people. The software can be better understood, and consequently better designed, because it can be understood one module at a time.

Unfortunately, *protocol* modules entail performance costs. One source of overhead is control transfer between protocols. For example, if each protocol were implemented as a process, passing a message from one layer to the next would involve a context switch. The protocol frameworks described above all use an *upcall* structure [Cla85] in which layers interface to each other via function calls within a common address space. While much more efficient than a context switch, this still entails some overhead.

A less obvious but more significant source of performance cost is information hiding. The best criterion for the decomposition of software systems into modules is the hiding of design decisions [Par72]. Unfortunately, this has two potential pitfalls for performance. First, potentially useful global information may not be available to any of the protocol modules. For example, each protocol might have to test whether there is enough space left in a message data structure for the protocol to add its header. If instead all the protocols were combined into a single protocol with a single large header, this test would be performed only once, or perhaps avoided altogether by allocating sufficient header space. Second, one layer may have information that could be useful at another, but the second layer must do without, or perhaps recompute the information, because the information that can be passed between layers is restricted by an interface. For example, two adjacent protocols may both manipulate message data, iterating through the data in a message performing some computation. If one protocol knew what data manipulation the other needed to perform, it could combine it with its own data manipulation, eliminating redundant memory accesses and loop overhead.

Conventional network software is limited to coarse-grain modularity because of the performance penalties for layering. Performance costs discourage the hiding of design decisions in separate protocols. Instead, design decisions are combined in large, complex protocols that are hard to design, implement, debug, modify, and maintain.

Clark has argued for even less modularity in network software [Cla82, CT90]. Because protocol specifications leave flexible the exact nature of the interface between adjacent protocols, it is entirely feasible to combine the implementation of adjacent protocols into a single module, as long as their behavior is consistent with the layered specification. Clark has advocated the use of this technique to improve performance.

In theory, the performance penalties for layering should make highly modular network software slower than less modular software; in practice, however, highly modular network software has performed comparably with less modular software [HPAO89, OP92]. What this demonstrates is that the software development support provided by protocol frameworks buys enough performance to compensate for performance losses due to modularity. In other words, there are two main factors determining performance: the performance that could potentially be obtained given a particular degree of modularity, and the development support (increasing with modularity) that determines how close the programmer will come to an implementation that achieves the theoretical potential. Low-modularity software tends to fall far short of its potential due to the difficulty of developing the software; but highly modular, framework-supported software comes much closer to attaining its somewhat lower potential. In summary, modularity seems to pay back part of its own performance cost by contributing to better implementations.

Morpheus promotes modularity of network software by using constraints and compiler optimizations to reduce the performance penalty for protocol layering.

1.5.3 Software Reuse

Morpheus supports two forms of software reuse. The first is the reuse of system-provided software in the form of object code that a Morpheus compiler generates beyond the behavior explicitly specified by the programmer. This is the Morpheus equivalent of protocol framework utility routines, but makes up more of the low-level implementation of a protocol than can be supplied by utility routines. This reused software is a consequence of Morpheus's high level of abstraction; the higher the level of abstraction, the greater the portion of the executable implementation is implicitly provided by the compiler, and hence reused in different protocols.

Morpheus also supports reuse of individual protocol implementations. Morpheus protocol modules may be flexibly composed with different adjacent protocols in different contexts, allowing them to be reused in different protocol graphs. This is made possible by Morpheus's *uniform protocol interface* (UPI) and its support for a high degree of modularity.

Protocol reuse requires a UPI so that arbitrary protocols are syntactically composable. The x-kernel on which Morpheus is based imposes a UPI, but this interface admits a number of loopholes that interfere with syntactic composability. The Avoca UPI [OMa90] is a revision of the x-kernel UPI that increases the likelihood of syntactic composability. The Morpheus UPI is likewise a revision of the x-kernel UPI, in part to increase composability, but it incorporates different solutions to the x-kernel's composability problems.

Protocol reuse also requires a high degree of modularity. If a protocol module performs a combination of functionalities motivated by a particular context of adjacent protocols, that module is not likely to be appropriate in other contexts. If, on the other hand, a protocol module encapsulates a single, "atomic" function, then that module is more likely to be useful in other protocol graphs. As described above, Morpheus promotes modularity by using constraints and compiler optimizations to reduce the performance penalty for protocol layering.

Fine granularity of *reusable* modules has an additional requirement beyond a low performance penalty: the uniform interface must accommodate fine grain decomposition. If reusability were not a concern, each interface between modules could be customized to the particular decomposition; but where reusability is a requirement, all the modules have identical interfaces. The design of this interface determines the kinds of decompositions that are possible. For example, if the uniform protocol interface does not support the sharing of flow control information, then flow control cannot be encapsulated in separate protocol modules, and instead each protocol must implement its own flow control or do without. By accommodating sharing of flow and congestion control information, Morpheus's uniform protocol interface supports finer grain decomposition than the x-kernel or Avoca.

I refer to the decomposition of network software into simple, reusable protocol modules as the *building-blocks approach* to developing network software. Such protocol modules can be used as building-blocks, composed to implement the same communication services that might have been implemented using a few, large protocols. This approach has been advocated previously [HPAO89, OMa90, OP92], but Morpheus contributes new support through its reduction of the performance penalty for layering and its protocol interface.

1.5.4 Performance

From the point of view of supporting network software development, Morpheus's performance optimizations reduce the performance penalty for layering. Since the performance cost per module is less, it is practical to decompose network software into finer-grain modules. This increased modularity makes it easier to develop and reuse network software.

There is an equally valid alternative view of Morpheus's optimizations: that they simply improve performance. If applied to a given protocol design with a fixed number of protocol modules, they will result in improved performance over an unoptimized implementation.

In other words, the performance payoff of Morpheus's optimizations is like money; it can be invested in greater modularity, or it can be banked as a performance improvement.

1.5.5 Scope and Limitations

The research presented in this dissertation is exploratory. It explores the potential of some new strategies for supporting the development of network software. It does not address a neatly circumscribed problem or provide a complete solution. In order to better focus on the potential of Morpheus's strategies, tangential concerns have been left incomplete.

The Morpheus problem domain has been limited to the asynchronous, one-to-one (unicast) protocols. This is the primary class of network communication, and includes TCP, IP, and UDP, as well as the low-level protocols that underlie other varieties of communication service. Synchronous communication (such as Remote Procedure Call) and multicast communication, while clearly important, are not addressed in this research.

The design and implementation of Morpheus has been left incomplete wherever it is not directly related to support for protocols. Consequently, there is no compiler, no formal semantics, and no grammar. The focus of this research is not language design and implementation, but rather on protocol abstractions and protocol-oriented compiler optimizations. The syntax and semantics of Morpheus protocol abstractions are presented informally, and the feasibility and performance of a compiler, including the optimization techniques, is argued indirectly.

Morpheus is designed for uniprocessor execution. Multiprocessing might well motivate different protocol abstractions and performance optimizations.

The use of specification-level constraints effectively limits Morpheus to supporting future protocols. Future protocols may be specified within the new constraints, but it is too late to constrain existing protocols. The particular constraints imposed by Morpheus are identified in the course of this dissertation and summarized in Chapter 5.

1.6 Dissertation Overview

Chapter 2 shows how Morpheus provides a high level of abstraction. The Morpheus uniform protocol interface is presented in that chapter. Chapter 3 shows how Morpheus reduces the latency penalty for layering, and Chapter 4 shows how Morpheus reduces the throughput penalty for layering. Chapter 5 summarizes Morpheus's constraints and makes some concluding remarks.

CHAPTER 2

MORPHEUS PROTOCOL ABSTRACTIONS

Morpheus's protocol abstractions support protocol development in two ways. First, they provide a high level of abstraction. This supports protocol development by providing a seamless model for thinking about protocols and relieving the programmer of making and expressing low-level design decisions. Second, these abstractions present a uniform protocol interface well-suited to decomposition. This permits reuse of simple protocol modules, the building-blocks approach.

This chapter begins by presenting the protocol abstractions, which are represented as objects. The high level of these abstractions is demonstrated by comparison with their low-level implementation. The uniform protocol interface is presented in the course of describing the protocol abstractions. A case is then made for the feasibility of implementing a Morpheus compiler. Finally, the protocol interface is contrasted with that of the x-kernel Uniform Protocol Interface on which it is based. The Morpheus protocol interface is shown to have greater syntactic composability and support a greater degree of decomposition.

2.1 Morpheus Objects

Morpheus represents the fundamental protocol abstractions as objects. The Morpheus model of protocols partitions state information such that each action operates on a specific body of state information. Object-oriented programming fosters this way of thinking by packaging data together with related procedures.

Morpheus provides two kinds of pre-defined objects. Utility objects are instantiated directly to provide services of use to many protocols. Protocol component objects are refined by the programmer to derive objects specific to a given protocol.

It is irrelevant to this research whether or not programmers can also define their own completely new objects. In the examples that appear in this dissertation it will be assumed that Morpheus does not support user-defined objects. Rather, objects representing protocol abstractions are embedded in an otherwise C-like language.

2.1.1 Utility Objects

Morpheus provides utility objects to perform some common services for protocols. In protocol frameworks, such utilities are implemented as library routines. The utility objects are *Messages*, *Maps*, and *Events* [HMPT89].

Maps provide a generic mapping service, mapping values of one type into values of another (possibly identical) type. They provide operations for entering, looking up, and deleting mappings from one value to another. They are useful for mapping from one type of address into another. A Map is implemented as a hash table.

Events provide a mechanism for scheduling the future execution of a specified function. Events may be scheduled or cancelled. Protocols can use them to send periodic "I am alive" messages, or to take recovery actions if a message is not acknowledged within the expected interval, for example. Events are implemented in the Morpheus runtime system, using operating system timing support.

Messages are Morpheus's most interesting utility object due to several novel features. First, header fields are always word aligned, making access more efficient. This is made possible by a constraint on protocol specifications that requires that headers and data each be an integral number of words and that individual header fields be word aligned relative to the start of the header.

Second, byte ordering conversions for header fields are performed automatically. Byte ordering of data is handled differently; this is discussed in Chapter 4. The byte ordering supported by a host machine may not match the byte ordering that a protocol specifies for its message header. Morpheus simplifies programming and increases portability by transparently resolving the potential mismatch. The byte ordering specified by the protocol is explicitly declared in a Morpheus protocol program. Hence, a Morpheus compiler knows both the protocol's byte order and the byte order for the compiler's target machine, so it can generate the appropriate object code for accessing header fields.

The last novel aspect of Messages is the *segregated message* abstraction. Segregated messages expose some message structure that is not exposed by the conventional message abstraction. Segregated messages are motivated and discussed in Chapter 4.

2.1.2 Protocol Component Objects

The Morpheus programmer implements a protocol by refining built-in base classes, thereby deriving subclasses that are specific to the protocol, as illustrated in Figure 2.1. A subclass is derived from a base class by adding new state information (declaring additional instance variables) and extending the base class behavior (defining additional procedure code that augments the base class procedures). A protocol implementation consists of object subclasses rather than object instances because a protocol entity generally comprises multiple instances of its objects; each instance of a protocol (each entity) is made up of objects that are instances of that protocol's subclasses. Furthermore, there can be more than one instance of a given protocol within the protocol graph of a single host.

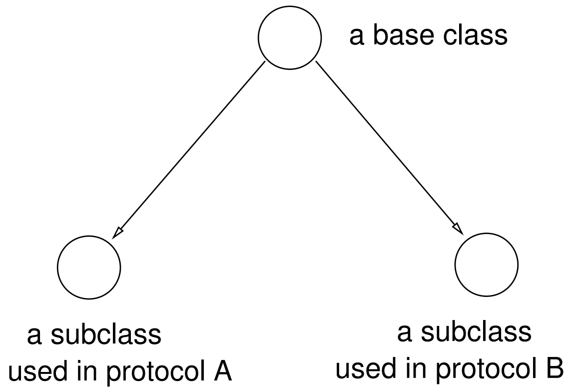


Figure 2.1: Protocols as Refinements

Morpheus defines base classes corresponding to the fundamental elements of Morpheus's model of protocols. These base classes—*Protocol*, *OverSap*, *UnderSap*, *OverSession*, and *UnderSession*—are schematically depicted in Figure 2.2. OverSaps and UnderSaps are components of Protocols, and OverSessions are components of OverSaps, while UnderSessions are components of UnderSaps.

A *protocol entity* is an instance of a protocol implementation. In Morpheus, a protocol entity is represented by a Protocol object.

A *Service Access Point (SAP)* is an interface between a communication service and a user of that service—it is the interface by which a service user accesses a communication service. The users of a communication service may be treated as protocols, and often are in fact higher level protocols implementing higher level communication services. Likewise, a communication service consists of a protocol at the top of a directed graph of lower level protocols. Thus, since a SAP is an interface between a communication service and a user of that service, it is also, more concretely, an interface between protocol entities. The using protocol is refered to as being *on top of*, or being *a higher level protocol of*, the the top protocol of the communication service. The top protocol of the communication

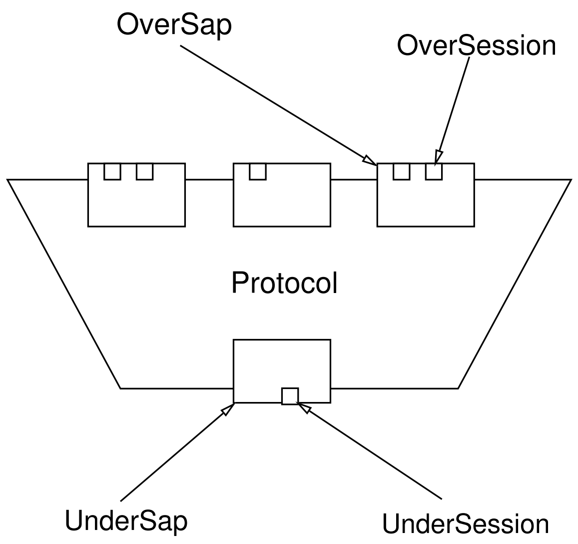


Figure 2.2: The Base Classes

service is refered to as *underlying*, or being *a lower level protocol of*, the using protocol.

An entity associates a unique address with each SAP to a higher level entity. There may be multiple such SAPs because an entity may serve more than one higher level entity. This address or *multiplexing key* associated with a SAP is used to tag messages so that outgoing messages from multiple higher level protocol entities can be multiplexed and arriving messages can be demultiplexed to the corresponding destination entities. An entity may also have multiple SAPs to lower level entities in order to use the different communication services (such as access to different local area networks) represented by the different lower level entities.

In Morpheus, a SAP is represented by a pair of objects, with one of the objects belonging to one of the involved Protocols, and one belonging to the other. An OverSap object represents a SAP shared with a higher level Protocol, and an UnderSap object represents a SAP shared with a lower level Protocol (an object is "over" or "under" with respect to the Protocol of which it is a component). The operations provided by an OverSap or UnderSap are invoked by the adjacent Protocol. For each OverSap or UnderSap object belonging to a Protocol, the other Protocol sharing the object has a corresponding object, an UnderSap or OverSap respectively, which provides operations invoked by the first Protocol. Thus a SAP, which is a two-way interface, is represented as an OverSap-UnderSap pair, as illustrated in Figure 2.3.

A *conversation* is the exchange of logically related messages between a pair of SAPs—

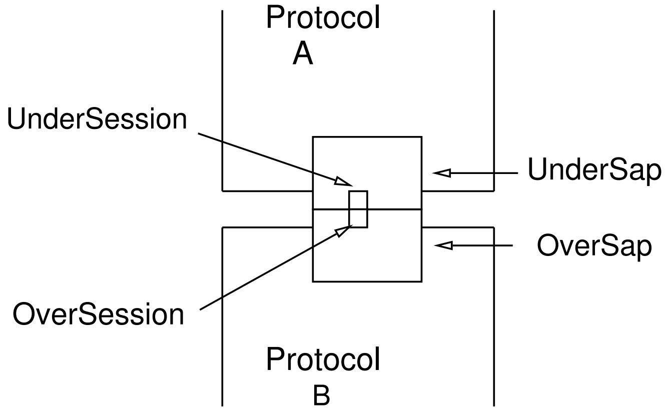


Figure 2.3: Sap and Session Objects

effectively, between a pair of peers. (In a multicast model, a conversation would involve a *group* of SAPs.) The interfaces to different conversations are treated as distinct from the SAP interface. The interface at one end of a particular conversation is a *session*. Operations on a session do not specify the pertinent conversation because that is implicit in the session. There are two addresses associated with a session: the address of the SAP at "this end" of the conversation, and the address of the SAP at "the other end."

There is an alternative to the session approach, which treats distinct conversations as having distinct interfaces. Instead, the sending and receiving of messages at one end of a conversation could be considered as part of the overall SAP interface. Operations involving a conversation would specify the pertinent conversation as an argument to the operation.

Morpheus takes the session approach, distinguishing the interfaces to different conversations. This is simpler and more efficient since it avoids specifying the pertinent conversation with every operation on the conversation. Since sessions, like SAPs, are two-way interfaces, Morpheus again uses a pair of objects to represent a session. OverSession-UnderSession pairs are illustrated in Figure 2.4.

The complexity of OverSap-UnderSap and OverSession-UnderSession pairs is concealed by Morpheus. For the programmer it is as if a pair were combined into a single object, with some of its operations implemented by one protocol and some implemented by the other. For example, instead of invoking an operation on a lower level protocol's OverSap, a protocol invokes the operation on its own corresponding UnderSap. The Over/Under distinction is maintained, however, since the operations a protocol provides

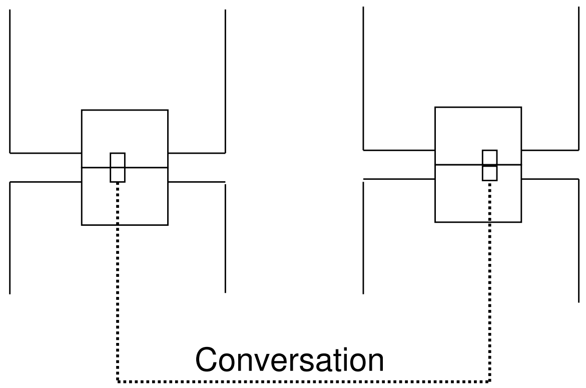


Figure 2.4: Sessions

for its OverSaps and OverSessions are different from those it provides for its UnderSaps and UnderSessions.

The operations supported by Morpheus objects make up the Morpheus protocol interface. These operations are not the same as the operations programmed by the object implementor. The operations supported by an object—the operations that one protocol invokes to get another protocol to do something—are called *external* operations. In contrast, *internal* operations are the operations programmed by the object implementor.

The gap between the external operations and the internal operations is both syntactic and semantic. There is a syntactic gap because one protocol has no way to invoke operations on, or even refer directly to, Sap or Session objects belonging to another protocol. Instead it invokes an operation on one of its own objects, which results in the operation being performed by the corresponding object with which it is transparently paired. For example, an external operation on an UnderSap would be realized as an internal operation on the corresponding OverSap.

There is also a semantic gap between some external and internal operations because the Morpheus implementation provides some of the semantics. The object implementor programs internal operations that are specific to a particular protocol, and the Morpheus compiler augments these internal operations with generic protocol behavior to make up complete external operations. A compiler can implement external operations in terms of the corresponding internal operations using a variety of techniques, including code generation, inheritance, and shared infrastructure routines.

The object operations are summarized in Table 2.1. For purposes of research, I have selected a minimal functional set of protocol operations; a practical system would require some additional operations, such as for terminating conversations.

EXTERNALLY INVOKED OPERATIONS	CORRESPONDING INTERNAL OPERATIONS
createProtocol(protocolClass,links)	protocol.addOverSap(overSap) protocol.initProtocol(underSaps)
underSap.getLocalAddr()	overSap.getLocalAddr()
underSap.enableUpwardSessionCreate()	overSap.enableUpwardSessionCreate()
underSap.createUnderSession(hostAddr)	*depends on the two shapes involved*
overSap.createOverSession(hostAddr)	*depends on the two shapes involved*
underSession.sendThruput(msg)	overSession.sendThruput(msg)
underSession.sendLatency(msg)	overSession.sendLatency(msg)
overSession.deliverThruput(msg)	underSession.deliverThruput(msg)
overSession.deliverLatency(msg)	underSession.deliverLatency(msg)
overSession.grantSends(number)	underSession.grantSends(number)
underSession.grantDelivers(number)	overSession.grantDelivers(number)

Table 2.1: Object Operations

These operations perform the following activities:

createProtocol(protocolClass, links) This external operation is unique in that it is not an operation on an object, and it is not invoked by a protocol. It creates a new protocol entity that is an instance of the protocol implementation given by **protocolClass**. The **links** arguments provide information for creating one or more UnderSaps, as explained later in this chapter. An integral part of the identity of a protocol entity is the set of underlying communication services to which it is attached via its UnderSaps, since these (together with the protocol itself) determine the communication service provided by the protocol entity. This **createProtocol** operation is used repeatedly to create Protocols on top of previously created Protocols, thereby incrementally constructing the graph of protocols on a given system.

The internal operation **protocol.initProtocol(underSaps)** initializes the newly created Protocol. The internal operation **protocol.addOverSap(overSap)** adds a newly created OverSap (corresponding to the newly created Protocol) to the set of OverSaps belonging to an existing Protocol.

underSap.getLocalAddr() This external operation returns the address of the host on which this SAP resides. The corresponding internal operation **overSap.getLocalAddr()** implements the address lookup.

underSap.createUnderSession(hostAddr), overSap.createOverSession(hostAddr) These external operations create an OverSession-UnderSession pair. The pair represents the local end of a conversation between the SAP specified by **underSap** or **overSap** on which the operation is invoked, and the corresponding remote SAP on the host identified by **hostAddr**. The corresponding internal operations depend on the shapes of the protocol involved.

underSap.enableUpwardSessionCreate() This external operation grants permission to the underlying protocol to open a conversation by creating an OverSession-UnderSession pair between the invoker and the underlying protocol. The corresponding internal operation **overSap.enableUpwardSessionCreate()** simply records that permission was granted, in case a message arrives for the specified SAP. The underlying protocol would then be allowed to invoke **overSap.createOverSession(addr)**. This makes it possible for a remote peer to initiate a conversation. Without this permission, the local peer is not subject to uninvited messages; a remote peer must wait until the local peer initiates a conversation via **underSap.createUnderSession(addr)**.

underSession.sendThruput(msg), underSession.sendLatency(msg) There are two operations for sending messages because one is optimized for latency and the other for throughput, and the optimizations show through at the source code level. These external operations pass a message to a lower level protocol to be transmitted to the session at the remote end of the conversation whose local end is represented by **underSession**. The corresponding internal operations, **overSession.sendThruput(msg)** and **overSession.sendLatency(msg)** respectively, transmit the message in accordance with their protocol. The two operations are semantically identical, but differ in their syntax and implementation. **SendLatency** is structured as an ordinary function and optimized for latency. **SendThruput** is structured as a collection of functions that share some data structures; this structure provides the necessary hooks for Morpheus's throughput optimization. This situation is motivated and discussed in Chapters 3 and 4. The remainder of this chapter is presented as though there were a single unified **send** in order to simplify the exposition.

overSession.deliverThruput(msg), overSession.deliverLatency(msg) As was the case for **sendThruput** and **sendLatency**, there are two operations because one is optimized for latency and the other for throughput, and the optimizations show through at the source code level. These external operations deliver an arriving message to a higher level protocol. The message is part of a conversation whose local end is represented by **overSession**. Note that a lower protocol initiates delivery of a message to a higher protocol, rather than a higher protocol initiating reception of a message from a lower protocol. The corresponding internal operations, **underSession.deliverThruput(msg)** and **underSession.deliverLatency(msg)** respectively, accept delivery of the message, possibly delivering it in turn to a yet higher level protocol. This existence of two sets of delivery operations is motivated and discussed in Chapters 3 and 4. The remainder of this chapter is presented as though there were a single unified **deliver** in order to simplify the exposition.

overSession.grantSends(number) This external operation grants permission to a higher protocol to send some number of messages as part of the conversation represented by **overSession**. It is a primitive for implementing flow and congestion control, which is discussed in greater detail below. The corresponding internal operation **underSession.grantSends(number)** records or acts on this permission.

underSession.grantDelivers(number) This external operation grants permission to a lower protocol to deliver some number of messages as part of the conversation represented by **underSession**. It is a primitive for implementing flow and congestion control, which is discussed in greater detail below. The corresponding internal operation **overSession.grantDelivers(number)** records or acts on this permission.

2.2 Protocol Shapes

Morpheus supports three kinds, or *shapes*, of protocols. Morpheus partitions protocol functionality into three categories, and each shape provides functionality from just the corresponding category. For example, one shape is responsible for any multiplexing. Arbitrary protocol functionality is implemented by composing protocols of possibly different shapes. Shapes constitute a constraint on protocol specifications; protocol specifications are not allowed to mix functionality from more than one of the categories.

The benefit of shapes is that they are particularly effective in raising the level of abstraction. In more concrete terms, they make it possible for a Morpheus compiler to automatically supply more of the code and data structures that the programmer would otherwise have to specify. A protocol's shape is declared in its Morpheus program. This declaration gives Morpheus extra information about a protocol because a protocol's shape determines much of what the protocol will do, and the data structures it will need. In contrast, much less can be inferred about the structure of a protocol of unrestricted functionality.

Shape conveys so much information about a protocol because it captures several characteristics that are tied together. The simplest explanation is that the partition is based on "plumbing:" does the protocol support multiple higher level protocols, or just one, and does it use multiple lower level protocols, or just one? The three shapes, *multiplexor*, *router*, and *worker*, are schematically depicted in Figure 2.5.

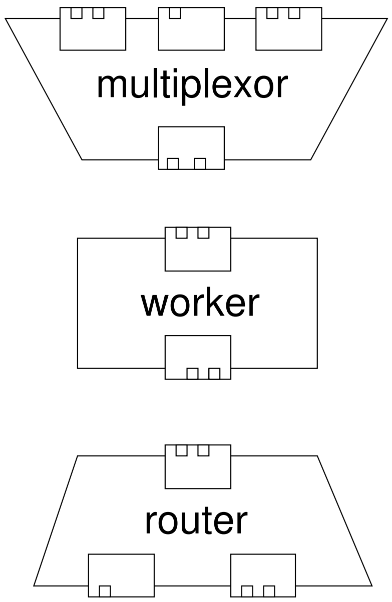


Figure 2.5: The Shapes

Multiplexor protocols multiplex messages being sent from different sessions, and demultiplex those messages to the corresponding sessions when they are delivered. *Router* protocols make runtime decisions regarding which lower level conversation (UnderSession) to use to send a message. The decision could be made on a per-message basis or a per-OverSession (higher level conversation) basis. Hence, Morpheus routers are more general than is usually suggested by the term "router" (e.g. IP); they determine not only the series of links that a message follows through a hardware network, but also the series of protocol entities that a message traverses within the protocol graph on a host. *Worker* protocols do what might be described as "the real work" such as error detection, buffering for retransmission, and detecting lost, reordered, or duplicated messages. In particular, any manipulations of message data are performed by workers.

2.2.1 Worker Protocols

A worker protocol is essentially a message filter. It has just one higher level protocol and just one underlying protocol. The correspondence between OverSessions to the higher protocol and UnderSessions to the underlying protocol is one-to-one and fixed. Hence, a worker focuses on some message processing function without being encumbered by routing, multiplexing, or the processing of any sort of addressing information.

The code in Figure 2.6 is the Morpheus program for a worker protocol called SEQUENCER. SEQUENCER's function is to reject any duplicate or out-of-order packets. SEQUENCER's function does *not* include any guarantee that every message sent is delivered; in the building-blocks approach fostered by Morpheus, that would be the function of one or more other protocol layers.

```
Worker SEQUENCER                            /* protocol SEQUENCER has shape "worker" */

LittleEndian Header { unsigned seqNum; }     /* declare header format */
Protocol { unsigned sendSeqNum; }            /* declare Protocol state variables */
UnderSession { unsigned receiveSeqNum; }     /* declare UnderSession state variables */

/* no programmer-declared state variables needed for the other classes */

initProtocol( underSaps ) { sendSeqNum = 1; }

initUnderSession() { receiveSeqNum = 0; }

send(msg)
{
    /* header prepended implicitly */
    msg.hdr.seqNum = sendSeqNum++;
    underSession.send(msg);                  /* underSession: inherited state variable */
}

deliver(msg)
{
    if(msg.hdr.seqNum > receiveSeqNum){
        receiveSeqNum = msg.hdr.seqNum;
        /* header stripped implicitly */
        overSession.deliver(msg);            /* overSession: inherited state variable */
    }
    }else
        grantDelivers( 1 );                  /* restore 1 credit since message dropped */
}
```

Figure 2.6: A worker protocol program

The primary point of this example is that SEQUENCER's Morpheus program consists exclusively of information that is specific to SEQUENCER. In contrast, the implementation of a protocol in a general purpose language inevitably involves considerable "boilerplate" code that is the same for many or all protocols, because the general purpose language does not know about protocols. For example, consider the C implementation of SEQUENCER in Appendix A. It includes data structure declarations and code for creating and assembling the component objects, connecting SEQUENCER to the adjacent protocol layers, creating conversations, and adding and deleting message headers. Using Morpheus, these routine aspects of a worker protocol are all implemented implicitly. The

SEQUENCER Morpheus program is succinct because one need express only those design choices that are specific to SEQUENCER, not those that can be made in advance for arbitrary worker protocols.

Morpheus implicitly provides data structures such as the state variables **underSession** and **overSession**. To understand what they represent and why they should be implicitly provided, consider the nature of a worker. Since a worker does no routing or multiplexing, there is a fixed, one-to-one correspondence between OverSessions to the higher protocol and UnderSessions to the underlying protocol. Hence each OverSession uses the state variable **underSession** to identify its corresponding UnderSession for use in relaying a message; and similarly in the reverse direction.

Morpheus also implicitly provides code, or *behavior*, such as the initialization of the aforementioned state variables **overSession** and **underSession**. Morpheus provides other data structures which are not explicit in worker protocol programs because they are used exclusively by implicitly provided code.

SEQUENCER incidentally illustrates how message header byte order is specified. The keyword **LittleEndian** indicates the byte order with which fields in the header (**seqNum** in this case) are to be represented. The Morpheus compiler uses this information to generate the appropriate code for accessing header fields, even though they might use a byte order different from the native byte order of the host machine.

2.2.2 Multiplexor Protocols

A *Multiplexor* protocol implements the sharing of conversations. A multiplexor supports a variable number of higher level protocols, represented by OverSaps, but uses just one underlying protocol, represented by an UnderSap. It provides a potentially large number of conversations from higher level protocols by combining, or *multiplexing* them in conversations provided by the underlying protocol. Incoming messages are *demultiplexed*, or separated into the appropriate higher level conversations, on the basis of the messages' headers. A pair of *multiplexing keys* in the header identify each message's source and destination. The source multiplexing key identifies the source SAP relative to its underlying multiplexor entity, and the destination multiplexing key identifies the destination SAP relative to *its* underlying multiplexor entity.

Since multiplexors implement the sharing of conversations, they must implement a *policy* for sharing conversations. This amounts to a policy for the scheduling of outgoing messages from competing higher level conversations. This is the dimension along which multiplexors vary. The simplest multiplexor transmits messages first-come-first-serve.

More sophisticated multiplexors transmit messages in an order based on priority or quality of service considerations, as permitted by flow control.

The Morpheus program for a multiplexor protocol expresses only the policy for scheduling outgoing messages; Morpheus implicitly provides the rest of the implementation. This is illustrated by the multiplexor FCFS shown in Figure 2.7. FCFS stands for First-Come-First-Serve, the simplest policy; when a message is passed to FCFS via **send**, it passes the message directly to the underlying protocol via **send**. Morpheus provides FCFS with an implementation of the basic tasks performed by every multiplexor such as creating connections, appending message headers that identify the source and destination of a message, and demultiplexing messages based on their headers. For comparison, one C implementation of FCFS included over 180 lines of C source code, despite calling a variety of library routines.

```
Multiplexor FCFS                          /* protocol FCFS has shape "multiplexor" */

send(msg)
{
    underSession.send(msg);
}
```

Figure 2.7: A multiplexor protocol program

One reason that Morpheus is able to provide so much of a multiplexor's implementation is that a multiplexor's function is narrowly defined. Multiplexors vary in their scheduling of outgoing messages, but otherwise they all do the same thing.

The other reason that Morpheus is able to provide so much of a multiplexor's implementation is that it eliminates gratuitous design alternatives by imposing constraints on multiplexor protocol specifications. Potentially, each multiplexor protocol could use different types for its multiplexing keys. Morpheus mandates a single, universal multiplexor key type. Again potentially, a multiplexor could use a single multiplexing key that must be the same for both source and destination. This is sufficient in the case where all the peers that make up a protocol layer can use the identical multiplexing key. Morpheus mandates independent source and destination multiplexing keys, which covers the single multiplexing key case as a degenerate case. Since the multiplexing key type is the same for every multiplexor, and all multiplexors have independent source and destination keys, it is easy for a Morpheus compiler to generate the object code that deals with multiplexing keys.

In Morpheus, the multiplexing key values used to identify specific SAPs are determined when the protocols are composed into a graph; the key values are not hard-coded into any protocol. When a higher level protocol is composed with a multiplexor, the SAP linking them is labeled with the corresponding multiplexing keys. The keys are selected by neither the higher level protocol nor the multiplexor; rather they are selected by whatever software commanded the composition. The composition is specified by a command

createProtocol(protocolClass, link_1, ... link_n)

Each link argument identifies a protocol entity and, if that entity is a multiplexor, a pair of multiplexing keys. The information in each link is used to create an UnderSap for the newly created protocol. The form of a link argument depends on the underlying protocol's shape. If the underlying protocol is a multiplexor, then the link argument is a tuple of the form (theMultiplexorEntity, key1, key2). If the underlying protocol is a worker or router, the link argument consists solely of the underlying entity, without any keys. Hence multiplexing keys are specified as part of the act of creating/composing the protocol entities.

Morpheus is at odds with more traditional protocol models that assume that multiplexing is a basic part of every protocol. There are two strong justifications for dropping this assumption. First, Morpheus is intended to support simple building-block protocols. Functionality that would have been combined in a single conventional protocol is instead *decomposed* into a collection of Morpheus protocols. To the extent that a single level of multiplexing is appropriate for a conventional protocol, the equivalent collection of Morpheus protocols need provide only a single level of multiplexing.

The second reason that multiplexing is not a basic part of every Morpheus protocol is that layered multiplexing is "considered harmful" [Fel90, Ten89]. One level of multiplexing per host is required to share the network hardware. *Logical* or *layered* multiplexing at additional levels in the protocol graph is not strictly necessary and has significant disadvantages, among them:

- Conversations that have been merged cannot be distinguished for purposes of quality of service.
- Multiplexing at multiple layers hurts performance by duplicating effort.
- Multiplexing is a barrier to the propagation of flow and congestion control information between protocol layers.

- Multiplexing complicates application of optimizations based on Integrated Layer Processing [CT90].

Morpheus does not assume any logical multiplexing; a Morpheus protocol graph may have a single multiplexor at the bottom. Furthermore, Morpheus's inclusion of a flow control interface between layers is predicated on the assumption that relatively few layers multiplex.

2.2.3 Router Protocols

Router protocols use multiple underlying communication services and make runtime decisions regarding which one should be used to transmit a given message. The decision may depend on either the individual message or the higher level conversation. Hence, Morpheus routers are more general than is usually suggested by the term "router" (e.g. IP), in that it includes not only determining a path through the hosts on a hardware network, but also determining a path through the protocol graph on a host, as illustrated in Figure 2.8.

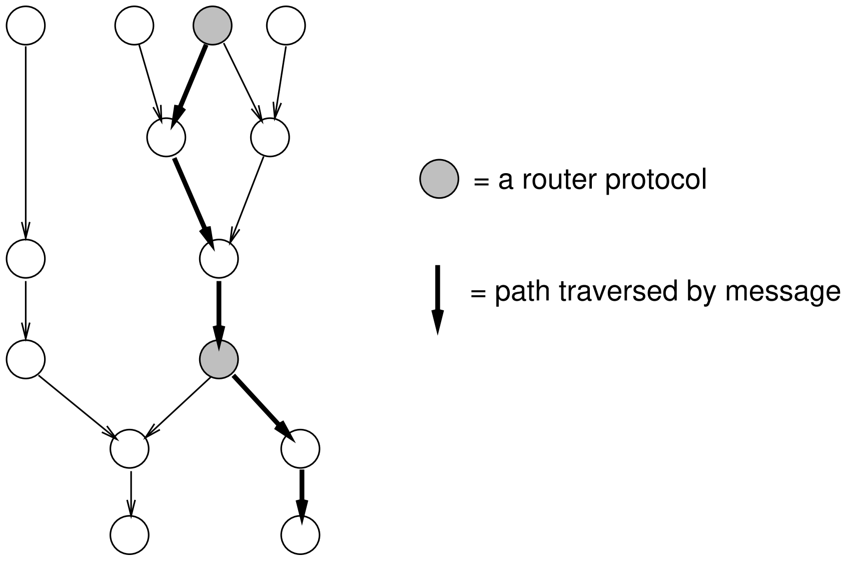


Figure 2.8: Routers in a Protocol Graph

Router protocols can vary drastically, limiting the part of their implementation that Morpheus can implicitly provide. Routers can vary even in their plumbing. For example, the relationship between OverSessions and UnderSessions depends entirely on the particular router. Since routers vary so significantly, Morpheus can predict only relatively little of a router's structure; routers are the shape for which the programmer must specify the most information.

The code in Figure 2.9 is the Morpheus program for a router protocol called SIZER. SIZER is so called because it uses the size of each outgoing message to determine which of two underlying communication services to use. SIZER could be used to build a protocol graph in which messages requiring fragmentation and reassembly follow one path through the graph (e.g., one which includes a fragmentation/reassembly protocol), while smaller messages follow a different path, with the two paths rejoining via a multiplexor at a lower level.

```
Router SIZER

/* SIZER is a "virtual protocol," i.e. it has no header */

Protocol{
    UnderSap      smallUnderSap;
    UnderSap      bigUnderSap;
};

OverSession{
    UnderSessn    smallUnderSessn;
    UnderSessn    bigUnderSessn;
};

#define   MAX_SMALL_MSG_SZ  1000  /* max msg size accepted by small srvc*/

send( msg ){
    if( msg.len <= MAX_SMALL_MSG_SZ ){
        smallUnderSessn.send( msg );
    }else{
        bigUnderSessn.send( msg );
    }
}

deliver( msg ){
    overSession.deliver( msg );
}

initProtocol( underSaps ){
    smallUnderSap = underSaps[0];
    bigUnderSap = underSaps[1];
}

getLocalAddr(){
    /* assumes both small and big srvcs use identical addr space */
    return( bigUndrSap.getLocalAd() );
}
```

```
enableUpwardSessionCreate(){
    bigUnderSap.enableUpwardSessionCreate();
    smallUnderSap.enableUpwardSessionCreate();
}

initOverSessionDown( addr ){
    bigUnderSessn = bigUnderSap.createUnderSession( addr );
    bigUnderSessn.overSessn = self;
    smallUnderSessn = smallUnderSap.createUnderSession( addr );
    smallUnderSessn.overSessn = self;
}

initUnderSessionUp( addr ){
    overSessn = overSap.createOverSession( addr );
    if( underSap == bigUnderSap ){
        overSessn.bigUnderSessn = self;
        overSessn.smallUnderSessn = smallUnderSap.createUnderSession( addr );
        overSessn.smallUnderSessn.overSessn = overSessn;
    }else{
        overSessn.smallUnderSessn = self;
        overSessn.bigUnderSessn = bigUnderSap.createUnderSession( addr );
        overSessn.bigUnderSessn.overSessn = overSessn;
    }
}
```

Figure 2.9: A Router Protocol Program

initOverSessionDown and **initUnderSessionUp** are internal operations provided by router protocols corresponding to the external operations **underSap.createUnderSession** and **overSap.createOverSession** respectively. **initOverSessionDown** is an OverSession internal operation that is executed when a higher level protocol invokes the external operation **underSap.createUnderSession** on a SAP that it shares with SIZER. **initUnderSessionUp** is an UnderSession internal operation that is executed when a lower level protocol invokes the external operation **overSap.createOverSession** on a SAP that it shares with SIZER.

In contrast with previous examples, SIZER is dominated by plumbing code. The plumbing for workers and multiplexors is more constrained, and hence more of it is provided by Morpheus.

In general, a router protocol has to define its own space of host addresses (more accurately, a space of identifiers for the router and its peer entities) since the space of host addresses provided by the underlying communication services may differ. SIZER is an exception because it is designed for the case where the two underlying communication services use the same addresses, and so SIZER can let the underlying space of host addresses show through as its own. A router that defines its own space of host addresses must know how to translate between its space and those implemented by the underlying communication services, so that it knows which underlying service to use to reach a destination host address. Knowing how to translate addresses is also necessary for a router entity to learn its own identity; it uses **getLocalAddr** to determine its address with respect to an underlying communication service, which it must then translate into an address with respect to its own space.

To the extent that routers incorporate specific addressing information, they are less reusable and more specific to a particular composition of protocols than are the other shapes. Workers and multiplexors do not incorporate any specific addressing information internally. Their only interaction with host addresses is to relay **getLocalAddr** requests to the next lower layer until it reaches a router or the software that interfaces between the Morpheus protocol subsystem and the underlying network hardware. Multiplexing keys are not hard-coded into protocol code, but instead specified externally when protocols are composed. In contrast, routers should perhaps be regarded as part of the information that determines the composition of a protocol graph; they have to incorporate information about the host address spaces of the protocols with which they are composed.

2.3 Flow and Congestion Control

Flow and congestion control are common protocol functions. Flow control is responsible for ensuring that a protocol entity does not transmit messages to a peer faster than the destination peer can process the incoming messages; flow control synchronizes the sender with the receiver. One common approach to flow control involves a sliding window, in which the receiving peer indicates, in the form of a window of message sequence numbers, how many messages it is currently willing to accept from the sender. Congestion control is responsible for avoiding situations in which the capacity of one hardware link is exceeded by the aggregate message traffic from a collection of conversations that are all routed over the same link. Congestion control can involve routing decisions as well as the throttling of message transmission. The information on which it is based can take many forms, for example the rate at which messages are lost, since this is most often due to congestion.

If all protocols in a protocol graph multiplex, then each protocol must implement its own flow and congestion control or go without. The information on which control is based cannot be shared between layers because control information applies to individual conversations, and multiplexing combines multiple conversations into one (when sending) and separates one conversation into many (when delivering).

In contrast, wherever layers do not multiplex, there is the potential to encapsulate flow or congestion control in a separate protocol module because the same conversations that flow through that protocol also flow through some number of adjacent protocols.

Morpheus supports the encapsulation of flow and congestion control in modules separate from other protocol functionality by providing a flow and congestion control interface. The interface consists of the **underSession.grantDelivers(numberOfMessages)** and **overSession.grantSends(numberOfMessages)** operations. **grantSends** propagates information to the next higher level protocol, and **grantDelivers** to the next lower level protocol. Both operations express control information in terms of permission to pass the specified number of messages. Although control information is expressed in terms of permission to pass messages, any enforcement is implemented by protocols, not by the Morpheus language. In other words, Morpheus provides the *mechanism* for sharing flow and congestion control information, but the flow and congestion control *policies* are determined by the particular protocols. Different policies are appropriate in different circumstances, and at different points in a protocol graph. Furthermore, a protocol's **send** policy may well differ from its **deliver** policy. The Morpheus approach is to implement the policies as protocols, and reuse the protocols wherever the policies are appropriate.

Policies fall into three general categories. The first two categories include policies used by protocols that are not actively involved in flow or congestion control. The first category consists of a single policy: the bypass policy. In this case, the protocol simply relays the flow control information to the next protocol, and relies on that protocol to comply. This is likely to be appropriate for protocols where there is a one-to-one relationship between messages passed to the protocol and messages it passes on. The second category of policies likewise consists of a single policy: the "no flow control" (or "infinite credit") policy. Under this policy, the protocol behaves as if it had infinite credit to pass messages. Furthermore, it does not relay flow control information, nor does it expect to receive such information. This policy makes sense for a protocol that relies on a subsequent protocol to enforce flow control, e.g. by dropping messages when credit is unavailable.

The third category of policies includes all the "real" policies: those that receive flow control information, and comply with it by either enforcing control on other layers or propagating their own flow control information to other layers. Since Morpheus protocols are decomposed into the simplest possible elements, a protocol whose policy falls in this category should either have flow control as its sole function, or have a primary function with which flow control is inherently intertwined, such as horizontal flow control, or multiplexing on the basis of quality of service.

In conventional protocols, flow control is combined with many other functions in a single protocol, and it provides synchronization between peer instances of only that protocol. In Morpheus, a flow control protocol should perform no other function, and it should propagate flow control information to provide flow control between peers in higher layers. Suppose a protocol entity needs to control the flow of messages from a peer. It uses **grantDelivers** to throttle the delivery of messages from lower layers. This information propagates down to a flow control protocol, which translates the information into a message that propagates the information to its peer. The flow control protocol peer then uses **grantSends** to propagate the information up to the peer of the original protocol. Since this peer, the source of the messages, sends messages only after first receiving credit to do so, flow control is achieved. Figure 2.10 shows the pertinent routines from an example flow control worker protocol. This protocol is designed for the case where the underlying service is reliable, so control information is transmitted in the form of a number of message credits (as opposed to, for example, a window of sequence numbers).

The vertical propagation of flow control in either direction cannot continue through a multiplexing layer. This is because flow control information applies to one data stream, and a multiplexor always combines multiple streams into one (when sending) or separates one stream into many (when delivering). Therefore a multiplexor must in effect grant infinite credit in both directions; that is, the adjacent layers should assume that they have infinite credit to pass messages to the multiplexor. Although a multiplexor does not propagate flow control, it is essential that the multiplexor be informed of send credit. This allows the multiplexor to block or discard sent messages when credit is lacking, and resume blocked sends, distinguishing between messages on the basis of quality of service requirements.

A multiplexor could comply with flow control information regarding *deliveries* by blocking threads or dropping messages. This has the major drawback that all of the component streams get the same delivery flow control policy. A better approach is for each of the component streams to have its own delivery flow control policy implemented at higher levels, while the multiplexor applies the "infinite credit" policy to deliveries. This has the additional advantage of decoupling the deliver policy protocol from the send policy protocol, so that they may be varied independently by composing different protocols.

```
deliv( msg )
{
        if( msg.hdr.msgKind == DATA_MSG ){
            /* just pass it on up */
            overSessn.deliv( msg );
        }else if( msg.hdr.msgKind == CREDIT_MSG ){
            /* propagate the credits as vertical flow control info */
            overSessn.grantSends( msg.hdr.numCredits );
            msg.destroy();
        }else if( msg.hdr.msgKind == OPEN_MSG ){
            /* msg already completed its mission
            * when it caused sessns to be created
            */
            msg.destroy();
        }
}

grantDelivs( numCredits )
{
Msg             msg;

        /* create and send a credit msg */
        msg.create();
        msg.hdr.msgKind = CREDIT_MSG;
        msg.hdr.numCredits = numCredits;
        undrSessn.send( msg );
        msg.destroy();
}
```

Figure 2.10: Flow Control Protocol Fragments

2.4 Feasibility of a Morpheus Compiler

I have not implemented a Morpheus compiler because that would involve a great deal of effort only indirectly related to my thesis. The focus of this research is not language design and implementation, but rather protocol abstractions and protocol-oriented compiler optimizations.

However, Morpheus's protocol abstractions have naturally been designed to be compilable. Compilability essentially means that there is enough information in a program for a compiler to generate an object code implementation. Morpheus has been designed to provide just enough expressiveness for the programmer to provide the information that is unique to a given protocol. That information is sufficient, when combined with a compiler's built-in information about Morpheus's constrained variety of protocols, to produce a low level implementation.

The gap between Morpheus source code and its object code implementation reflects

Morpheus's high level of abstraction, and arguments regarding the feasibility of implementing the abstractions have been presented as they were introduced. To recapitulate, the parts of a low-level, object code implementation of a Morpheus protocol that are not *explicitly* specified in the Morpheus source code are nonetheless determined, by the following means:

Constraints. Constraints, both at the specification level and below, make it possible to predetermine specific implementations for many features, regardless of the particular protocol.

Shape. Implementations can be predetermined for the features that are characteristic of each shape, and each protocol program begins by declaring its shape.

Other declarations. Declarations such as the message header byte order result in additional object code that is not explicit in the Morpheus source code.

Morpheus's design was guided by a non-compiler prototype implementation. First, the Morpheus protocol interface was implemented in a protocol framework called x-prime, derived from the x-kernel. Support for shapes was then added in the form of source code templates, with one template for each shape. Each template consisted of C source code appropriate for the corresponding shape, except that a number of references to undefined macros appeared in the text. Programming a protocol consisted of defining the macros with data structures and code corresponding to the particular protocol. A template and a particular set of macro definitions could then be compiled into the object code for a protocol.

A Morpheus compiler could parallel this technique. The source program would be analogous to the macros, but instead of explicit macro definitions, parsing allows the information to be expressed in a more intuitive and flexible syntax. The compiler's code generation routines would be analogous to the templates and framework infrastructure, completing implementation details not specific to a particular protocol. A compiler only adds more options to the way in which various protocol features can be realized: conventional code generation, shared routines, or even object inheritance.

The protocol-framework-and-templates approach can duplicate some Morpheus features but falls short of others. Its syntax is crude and inflexible. It cannot mix programmer-specified code with implicitly provided code at a fine granularity. Finally, it cannot use compiler optimizations to reduce the the performance penalty for layering. Reducing this penalty is important for not only modularity, and through it reusability, but also abstraction, since Morpheus's high level of abstraction depends in part on shapes, which in turn require a high degree of decomposition.

Another approach that can duplicate some Morpheus features but falls short of others is adding predefined object classes to a general purpose object oriented language (OOL) such as C++ [Str86]. In this approach, an OOL would be augmented with a collection of predefined object classes for Morpheus objects, and subclasses of those classes for each shape. However, like frameworks, general purpose OOLs would lack compiler optimizations that reduce the performance penalty for layering. Since the predefined classes would be written in the OOL source language just like any other classes, they would be unable to perform actions below the source language level transparently. Since the predefined classes would use the same general-purpose inheritance mechanism used for any classes, they would be unable to support fine-granularity mixing of inherited code with programmer-supplied code.

2.5 Comparison with the x-kernel Uniform Protocol Interface

Morpheus's protocol abstractions are descended from the x-kernel protocol framework, which has the basic goal of facilitating the development of high performance protocols. Thus, Morpheus has a second-generation model of protocols, based on experience with the x-kernel.

The Morpheus Uniform Protocol Interface supports a greater degree of decomposition than the x-kernel UPI due to Morpheus's flow control interface, which permits flow and congestion control to be encapsulated in their own protocol modules.

The Morpheus UPI also eliminates some of the x-kernel UPI's barriers to syntactic composability. These are the x-kernel's control operations and multiplexing scheme.

2.5.1 Control Operations

The x-kernel UPI's control operation is an escape hatch, like Unix's ioctl, that allows unrestricted interaction between protocols. One of its arguments is an opcode that identifies the true operation, and another argument is a pointer to a buffer in which a request or reply of arbitrary type can be passed. The control operation's purpose is to permit protocol operations that are supported by some but not all protocols.

The Morpheus protocol interface does not include a control operation because control operations limit syntactic composability. The problem is that a protocol that uses a particular control operation can only be composed with protocols that implement that particular control operation.

One use of the control operation is to set protocol options. This allows a single protocol entity to provide different communication services. For example, the User Datagram Protocol, UDP, can checksum the contents of its messages, or not, depending on an option. Requiring a single implementation to provide multiple services complicates the implementation and can adversely impact its performance. Each of the communication services could be implemented more simply by its own protocol.

Morpheus prohibits options. This constitutes a constraint on protocol specifications. Morpheus takes the position that, instead of one protocol with options, there should be a distinct protocol for each value of the options—a different communication service is implemented by a different protocol. The sharing of code between closely related variants of a protocol should be managed at compile time, not implemented by sharing object code at runtime. In those cases where the choice of appropriate variant depends on runtime information, a router protocol can be used to select the appropriate protocol on a per-session basis at conversation open time, or a per-message basis at send time.

Morpheus also eliminates the use of the x-kernel's control operation to learn the maximum packet size supported by the underlying communication service. This operation is used by x-kernel fragmentation/reassembly protocols to determine the size of the fragments into which outgoing messages must be fragmented. This situation is like protocol options in that it can be resolved at compile time, with each value of the maximum packet size made a constant in a distinct protocol.

Morpheus eliminates another common use of the x-kernel control operation, that of learning the addresses of the two ends of a conversation. In Morpheus, the address of the local end of a conversation can be learned by invoking the explicit **getLocalAddr** operation. The address of the remote end of a conversation is reported to a protocol when the conversation is created, and is easily recorded in a programmer-defined state variable in the corresponding UnderSession, if needed. In the x-kernel, a session is represented by a single (informal) object that is a component of the lower level protocol, so the higher level protocol does not have a convenient way to record the remote address.

Morpheus increases composability by eliminating control operations, but Avoca [OMa90] uses a different approach called *inherited controls*. Avoca provides a control operation interface, and Avoca protocols are responsible for implementing a control operation that, depending on the opcode of the particular operation, either performs the itself, if possible, or in turn invokes the same control operation on lower level protocols.

Inherited controls retain the flexibility of control operations while increasing the likelihood that a composition of protocols will be compatible, since the control operation used by one need not be provided by an immediately adjacent lower level protocol. However, this is still not as composable as Morpheus since there must still be *some* lower level protocol that implements the control operation. Also, inherited controls require a mechanism for assigning globally unique opcodes, so that control operations are not misinterpreted by intervening protocols.

2.5.2 Multiplexing

Multiplexing involves the assignment of identifiers called *multiplexing keys* to SAPs. A multiplexor marks each message with source and destination SAP multiplexing keys, thereby identifying the higher level conversation of which the message is a part.[1] This identification is the basis for demultiplexing received messages to the appropriate higher level conversations. Hence, a multiplexor must know the multiplexing keys that identify each of the higher level conversations it supports. There are various schemes by which a multiplexor might come to know these keys.

In the x-kernel design on which Morpheus is based (a newer x-kernel design is discussed below), multiplexors learn keys by a scheme which limits the reusability of protocols. The multiplexing key identifying a SAP is hardwired into the higher level protocol, which it communicates to the underlying multiplexor when it opens a conversation. This multiplexing key must be of a type determined by the multiplexor, and must be different from all other keys used by the multiplexor. Hence the higher level protocol cannot be reused in situations where the key type or assignment of keys to protocols is different.

Morpheus avoids this problem by explicitly associating the multiplexing key with the SAP that connects the two protocols, rather than having one protocol communicate it to the other. The assignment of a key to a SAP is made in the composition specification, not in any protocol. Each key is stored as state in the corresponding OverSap, so a multiplexor learns multiplexing keys for each SAP when the SAPs are created and installed.

This addressing scheme is sufficient to increase composability by eliminating multiplexing key mismatches, but Morpheus goes further to simplify multiplexing. Some protocols, such as IP, identify both ends of a conversation using the same multiplexing key—in effect, limiting protocols to conversations with other instances of themselves.

[1] Some non-Morpheus multiplexors require the SAPS at both ends of a conversation to have identical multiplexing keys. Their use of a single value to identify both ends of a conversation is a special case.

Other protocols, such as TCP, require that multiplexing keys be specified for both ends of a conversation, allowing clients to communicate with servers, for example. Morpheus imposes the constraint that all multiplexors require distinct keys (with possibly identical values) to be specified for each end of a conversation. This accommodates the shared key situation as a special case, and ensures greater uniformity of Morpheus multiplexors. Morpheus also constrains all multiplexors to use the same data type for multiplexing keys, again for uniformity. These constraints contribute to the high level of abstraction of Morpheus's multiplexor shape, by allowing Morpheus to implicitly provide more of a multiplexor's implementation with a minimum of programmer specification.

Avoca [OMa90] approaches the composability problems of x-kernel addressing in a different way. Avoca assigns each protocol a globally unique multiplexing key which it uses regardless of the underlying multiplexor. This eliminates the problems of mismatched key types, since all keys have the same type, and key clashes (except for one situation), since keys are globally unique. All implementations of a given protocol specification share the same identifier. This approach has the awkward shortcoming that multiple instances of the same protocol cannot be composed on top of the same multiplexing protocol since their keys would be identical. This would seem to support my contention that multiplexing keys are properly thought of as an attribute of the composition, not the protocol.

In Avoca, the higher level protocol is responsible for some of the work normally performed by the multiplexor. The higher level protocol marks outgoing messages with the destination multiplexing key and removes the key from incoming messages. (It is unclear which protocol is responsible for adding or removing source multiplexing keys.) This seems to offer negligible advantage since it could as well be performed by the multiplexor. Furthermore, it has the disadvantage that each protocol does the work of adding and removing multiplexing keys even if it is not on top of a multiplexing protocol.

I have pointed out two drawbacks of Avoca's multiplexing: multiple instances of the same protocol cannot be composed over a given multiplexor, and protocols do the work of adding and removing multiplexing keys even when they are not composed over a multiplexor. If it were assumed that every protocol multiplexes, then these drawbacks do not arise. However, this assumption limits decomposition into simple protocol modules, which is central to Avoca's (and Morpheus's) support for protocol development. Furthermore, as already discussed, there are strong arguments against layered multiplexing [Fel90, Ten89].

Avoca imposes a constraint on protocol specifications. It requires all protocol headers to begin with the destination multiplexing key and makes the higher level protocol responsible for adding and removing the key. Starting each header with its multiplexing key makes it possible to share that field with the multiplexor that uses it to demultiplex, a dubious benefit. This constraint allegedly serves the additional purpose of supporting protocol independent tools, such as a debugger perhaps; the protocol identifier is analogous to the return address in an activation record. Note however that without additional information, a tool would only be able to use the first such multiplexing key/protocol identifier in a message, because it would be unable to locate the protocol identifiers in higher level headers nested inside the message. A possible solution would be to constrain all protocols to also include a header length field at a fixed offset from the start of their headers. To continue the activation record analogy, this would be like a stack or frame pointer.

This constraint in Avoca is the only case of which I am aware, outside of Morpheus, of imposing specification-level constraints to support protocol development. However, Avoca does not use specification-level constraints as a general strategy.

A new version of the x-kernel, more recent than the design of Morpheus, uses a new multiplexing scheme. Configuration information supplies each protocol instance with a string identifier, although instances of the same protocol get the same identifier. A higher level protocol passes its identifier to a multiplexor. The multiplexor accesses other configuration information which maps the identifiers of the higher level protocol and the multiplexor into integer identifiers en route to mapping the pair into a multiplexing key, which is then cast into the appropriate type by the multiplexor.

The net effect is that the multiplexing key is determined by the composition specification, as in Morpheus. The indirection between protocol pairs and multiplexing keys is intended to support two coexisting strategies for assigning multiplexing keys: either an explicitly specified assignment, or, the default, the identify function applied to the higher level protocol's numeric identifier, as in Avoca. However, two anomalies follow from assigning the same string identifier to instances of the same protocol. First, there cannot be two instances of the same higher level protocol over the same multiplexor. Second, if there is more than one pair of instances of a given higher level protocol and multiplexor, then each such pair must use the same multiplexing key, even if the key is not the integer identifier of the higher level protocol.

In the x-kernel and in Avoca, the multiplexing key that identifies the remote end of a conversation is supplied, unless it can be inferred from the local key, by an application or high level protocol. The remote keys are passed to protocols along a path through the graph when the application or higher level protocol creates a conversation that will follow that path. The remote keys are not specified in the composition information; they may be hardwired into the protocols and application, or obtained from name services, or a combination of the two. In Morpheus, the multiplexing key that identifies the remote end of a conversation is also specified in the composition information and used to label the SAP. This is possible because within the protocol subsystem, the multiplexing keys by which a protocol identifies its destinations do not to vary across conversations; an entity always communicates with an identical peer or an opposite server/client aspect in the case of an asymmetric protocol. Greater flexibility is needed only at the application level, which is outside the protocol subsystem implemented by Morpheus. The interface between applications and the protocol subsystem is not specified in this research.

CHAPTER 3

LATENCY OPTIMIZATIONS

Reducing the performance penalty for protocol layers is a cornerstone of Morpheus's support for protocol development. Reducing this penalty makes fine-grain protocol modules practical. When performance penalties for layering are high, protocol developers are motivated to write large, complex, multi-shape, non-reusable protocols like conventional protocols.

Morpheus reduces the layering penalty by using compiler optimizations based on common patterns of protocol execution. Protocol-oriented optimizations would not be appropriate in general purpose languages, since those languages are intended for a much wider variety of programs that do not behave like protocols.

Latency and throughput are the coins in which layer penalties are paid. Unfortunately, Morpheus's throughput optimization tends to make latency worse. This seems to reflect a fundamental tension between latency and throughput. Morpheus resolves the competing demands of throughput and latency by splitting the message path in two, one path for throughput-dominant traffic and one for latency-dominant traffic. Messages are categorized as one or the other based on their length, since message length independent costs (latency) dominate for short messages, and message length dependent costs (throughput) dominate for long messages. Since messages tend to be either fairly short or fairly long, any reasonable choice of a threshold length for defining "short" versus "long" will work well.

The Morpheus programmer codes distinct operations for the latency-dominant and throughput-dominant paths. **Send** and **deliver** are split into two operations each: **sendLatency** and **deliverLatency** process short messages, and **sendThruput** and **deliverThruput** process long messages. The programmer must provide code for each path because the optimizations show through at the source code level. Chapter 4, which presents Morpheus's throughput optimization, concludes with a discussion of possible alternatives to the two-path scheme described in the preceding paragraphs.

The latency optimizations presented in this chapter all support the dynamic configuration of a protocol suite at runtime. Fixing the protocol suite before runtime would limit the extent to which communication services can be tailored to the needs of specific applications, since those applications arise at runtime. Furthermore, runtime configuration opens the possibility of runtime negotiations between a client and a server to select, from their local libraries of protocols, a set of protocols that both have available and that satisfies their joint communication service requirements.

This chapter presents the latency optimizations, reports experimental results regarding their effectiveness, and concludes with a discussion of alternative optimizations suitable for compile-time configuration.

3.1 Specific Techniques

There are five latency optimization techniques employed by Morpheus. The first three are compiler optimizations in the conventional sense. The fourth, while not a compiler optimization, is a direct consequence of using a domain-specific compiler. The fifth could be performed at the source code level of a general purpose language.

The latency optimizations are applied to **sendLatency** and **deliverLatency**. For clarity, the techniques are described in terms of **sendLatency**; they apply equally to **deliverLatency**.

Morpheus's latency optimizations are based on the common patterns of protocol execution. Consider the characteristics of the **sendLatency** operation. **SendLatency** takes a message as its argument. Since it is an operation on an OverSession object, there are in effect two arguments, the message and the OverSession. The typical **sendLatency** does some computation, accessing the object for state and other information, and using the built-in utilities; prepends a header to the message; and passes the message to the next lower layer via the **sendLatency** operation of another OverSession. This is repeated as the message passes through "many" layers. Morpheus optimizes for this common case.

Now consider how **sendLatency**s of adjacent layers interact at the object code level. **SendLatency** is implemented as a function at the object code level as well as the source code level. Morpheus protocols share the same address space, and hence interact via function calls. Function call conventions for modern RISC architectures are as follows[1]. The caller function places the calling arguments in registers designated for that purpose. If there are many arguments, the excess arguments are passed via the stack. The caller then executes a jump-to-subroutine instruction, which moves the return address into a designated register and transfers control to the callee function. The callee then updates the

[1]This assumes no register windows.

stack pointer to leave enough space on the stack for local variables, temporary variables, registers saved by the callee, and arguments to be passed to procedures called by the callee. Any registers that need to be saved, including the return address register, are then saved on the stack. By convention, certain registers (*callee save* registers) must have their contents saved and restored by the callee if it uses them; certain other registers (*caller save* registers) may be used freely, but must be saved and restored around a call site by the caller if they are to hold a live value across the call. In preparation for returning, the callee puts the result in a designated register. It then restores any saved registers, including the return address register, restores the stack pointer, and jumps to the return address.

3.1.1 Dedicated Message Registers

SendLatency's message parameter fits in a register because it is implemented as a pointer. If **sendLatency** calls any procedures, the message has to be saved so that another argument can be passed in the argument-passing registers (unless the called procedure takes the message as an argument, and in the same order in the argument list). Ultimately it must be restored to its original argument-passing register to be passed to the next layer's **sendLatency**. Morpheus modifies the parameter passing convention by setting aside a register specifically to pass the message. This register is selected from among the callee save registers. This way it is efficiently accessible in a register, and furthermore, that register need not be freed across subsequent calls to either the next layer's **sendLatency** or any other procedures.

The most heavily accessed part of a control message is its header. A pointer to the message header is used to access or modify fields in the header, and is incremented or decremented to prepend or strip headers. Morpheus optimizes for this by designating a callee save register for passing the header pointer explicitly along with the message object of which it is a part. This eliminates memory accesses otherwise necessary to read or write the header pointer state variable in the message object, and does so using a register that need not be saved across calls.

Message and header registers are initialized when the message is created, either to be sent or because it was just received via a network device. Also, the original contents of the two registers used are saved at that same time, and restored upon return. This overhead is amortized over the number of layers in the **sendLatency**, resulting in an insignificant per-layer cost. The message and header registers can be reallocated within a **sendLatency** if registers are in sufficiently short supply or if a second message must be passed, but this case is the exception. This optimization could be described as a second procedure calling convention that coexists with a primary calling convention.

All these implementation details are concealed from the Morpheus programmer, who sees only operations on a Message object.

3.1.2 Short-Circuit Return

Most often, the last action taken in a **sendLatency** is to invoke the next layer's **sendLatency**. When the lower **sendLatency** returns, the original **sendLatency** is done and also returns. Morpheus short-circuits such returns in a manner similar to optimizations for tail recursion, so that **sendLatency**s with no further work are bypassed in the sequence of procedure returns. Before calling the lower **sendLatency**, the current **sendLatency** restores all registers including the stack pointer. It then jumps to the lower **sendLatency**, but instead of giving a return address in the current **sendLatency**, it gives the return address provided by the current **sendLatency**'s *caller*.

This short-circuit return optimization in itself saves relatively little—a single jump assembler instruction per layer on a typical RISC processor. However, it contributes to another, conventional optimization that is more significant. If there are no procedure calls in a **sendLatency** operation, then that function can omit saving and restoring the return address register and updating and restoring the stack pointer. For this purpose, the short-circuit return effectively eliminates a procedure call. After applying short-circuiting, a significant number of **sendLatency** operations qualify as having no procedure calls. This occurs frequently since the typical Morpheus protocol is relatively simple.

This optimization is not implemented for general purpose languages because the benefit for the average general purpose program is small. In contrast, the Morpheus **sendLatency** and **deliverLatency** operations present a highly specialized domain, one that can be expected to benefit significantly from this optimization.

A variation on this optimization takes advantage of knowledge about the likelihood of executing various branches in object code that corresponds to a high level abstraction rather than being specified by a programmer. Suppose a procedure call were part of a branch that was known to be infrequently taken. Then instructions to manage the return address and stack pointer registers—i.e. a "lazy stack"—could be inserted just in that infrequent branch, so that they would be executed only if necessary.

3.1.3 Procedure Cloning

SendLatency nearly always accesses instance variables in its OverSession objects since these hold connection state information and other information such as the appropriate lower level UnderSession object. It also frequently accesses instance variables of the Sap and Protocol objects to which the Session object belongs. Most of the instance variables that are used internally are known to be constant because they have to do with connecting layers together, e.g. the OverSap corresponding to a UnderSap, or the source and destination host addresses in a multiplexor OverSession. User-declared instance variables are often constant as well.

Morpheus optimizes for this by generating a customized version of the **sendLatency** object code for each OverSession. At compile time, Morpheus generates a template for each protocol's **sendLatency**. When an OverSession object is created at runtime, a copy of the template is created and filled in—i.e. object code is modified—using the addresses of the Session, Sap, and Protocol objects and the values of those instance variables that are known to be constant. User-declared instance variables can be flagged as constant by a keyword. Chains of indirect pointers through memory are collapsed; for example, the address of the next layer's **sendLatency** replaces a chain of pointers that leads to it through the current layer's UnderSession and the next layer's OverSession. This also eliminates the need to pass the OverSession object as a parameter.

The end result of the technique is that constants are hardwired into the code. The constants could not be hardwired into an uncloned procedure because they are different for each clone. This hardwiring reduces the number of instructions executed for each clone, eliminating some memory accesses in the process.

This technique is a variation on *procedure cloning* [Coo83]. A procedure can be cloned to partition calls to it based on interprocedural constants information, or more generally, the solution to any forward interprocedural data-flow problem [Hal91]. Instead of a single procedure that must satisfy all calls, each clone is specialized to more efficiently handle its subset of the calls. The cloning practiced by Morpheus could not be arrived at by interprocedural analysis because the necessary information—the Session object for which the procedure is being cloned—is not available at compile time, since Sessions are created at runtime.

Morpheus' technique could also be classified as runtime code generation. The Synthesis kernel [PMI88] achieves exceptional performance using a similar technique. However, in contrast to Synthesis, which generates specialized kernel code, Morpheus generates specialized versions of protocol operations that are written by Morpheus programmers.

Morpheus' cloning has time and space costs. There is the time cost, paid at runtime, of making a copy of the template and filling in the appropriate constants. Although this occurs at runtime, it is part of communication channel creation—not in the time-critical **sendLatency** path. The space cost is an extra copy of the **sendLatency** code for each OverSession; that is, one for each communication channel currently provided by a protocol. There is already a space cost associated with each channel—a context-state. In Morpheus this is the OverSession object. The corresponding **sendLatency** clone could be considered a part of that state. Note also that each clone uses less space than an uncloned version of a procedure because of the simplifications enabled by the cloning. The increase in code space can be bounded by simply ceasing cloning once a code space threshold has been reached, as proposed in [Hal91]. This would require keeping one uncloned version of each **sendLatency** procedure to operate on any OverSessions that were not allocated their own clones.

The increased object code size due to cloning could conceivably have a negative effect on caching and virtual memory. Inlining results in a similar but greater increase in object code size, but inlining apparently has little effect on caching and virtual memory. [CHT91] found no obvious evidence of either thrashing or instruction cache overflow due to inlining, and cited previous reports of similar results. While these studies involved inlining, they suggest that increased object code size due to cloning would likewise be free of significant performance penalties.

3.1.4 Language Constructs for Frequent Tasks

Operations on Morpheus's built-in Message, Map, and Event objects are implemented as inline object code. This is more efficient than implementing this support in the form of a library of utility routines because procedure linkage code is eliminated and more context is exposed for conventional optimization. While similar results could be obtained using inline substitution of support routines (given a compiler which supported it), since these operations are language constructs in Morpheus, there is greater potential for optimization because the compiler has more information about the code being optimized. The costs of implementing support utilities as language constructs (as opposed to procedures) are increased compile time and increased object code size. These costs are held to reasonable limits in Morpheus because the set of utilities is fixed and small.

3.1.5 Eliminating Header Bounds Checking

The most frequent utility operations are pushing (prepending) and popping (stripping) headers. Although pushing a header usually amounts to incrementing a pointer, it can involve considerable bounds checking even in the case where no bounds are exceeded. Morpheus optimizes this away by allocating sufficient header space to each message as it is created, thereby ensuring that the header will not overflow. This is possible because the runtime system can determine the largest combined header that can possibly to prepended to a message based on the headers declared by the protocols in the current protocol graph.

3.2 Experimental Results

To study the impact of these latency optimizations in the absence of a compiler, I simulated generation of object code. This was accomplished by writing protocols in C according to the structure of Morpheus protocols; then compiling the C code using gcc into assembler language for the MIPS R3000 architecture; and finally applying the optimizations by hand at the assembler language level. I then performed two experiments to quantify the effect of Morpheus' optimization strategy: counting instructions and measuring end-to-end latency.

3.2.1 Instruction Counts

The effect of a given optimization depends on both the particular procedure and the other optimizations present. Therefore I have selected a particular protocol to use as an example, and report the effects as each optimization is applied in turn. The protocol is SEQUENCER, which was presented in Chapter 2. I focus on SEQUENCER's **sendLatency** operation. When SEQUENCER's **sendLatency** is invoked, it pushes a header onto the message. The header is filled in with a sequence number obtained from a Protocol state variable, which is then incremented. The message is then passed to the next protocol's **sendLatency**.

The results of the optimizations are summarized in Table 3.1. The first row of the table refers to the original, unoptimized version of the code, which consists of 45 assembler instructions. The final, optimized version consists of seven instructions.

Replacing the header push procedure with inline code reduces the common path by seven instructions—essentially the code for procedure linkage with the header push procedure. Eliminating header bounds checking eliminates an additional fifteen instructions. It also eliminates all conditional branches, so the common path is also the only path.

Dedicating registers for passing the message and its header eliminates an additional four instructions. This optimization generally gives a greater benefit in cases where there are procedure calls before calling the next layer's **sendLatency** (SEQUENCER has no

CUMULATIVE OPTIMIZATIONS	INSTRUCTIONS ELIMINATED	REMAINING INSTRUCTIONS
ORIGINAL VERSION	-	45
INLINE UTILITIES	7	38
ELIM BOUNDS CHECK	15	23
DEDICATED REGS	4	19
CLONING	7	12
SHORT-CIRCUIT	5	7

Table 3.1: Instruction Counts

such intermediate calls after applying the preceding optimizations); intermediate calls prohibit the message from remaining in an argument-passing register because that register is also used to pass arguments at the intermediate calls.

Cloning **sendLatency** eliminates another seven instructions. Several pointer indirections are short-circuited, and one less parameter is passed to the next **sendLatency** (i.e., its OverSession). Cloning and dedicated registers also each owe some of their benefit in this case to reducing by one the number of callee save registers needed.

Short-circuiting the return from the subsequent **sendLatency** results in the elimination of five more instructions. Short-circuiting the return makes it unnecessary to save the return address, which in turn makes it unnecessary to allocate stack storage.

The fully optimized SEQUENCER **sendLatency** consists of seven instructions: one to increment the header pointer, five to do "the real work" (increment the sequence number for outgoing messages and write it into the header of this message), and one to jump to the next layer. But not all assembler instructions are equal. Loads and stores can take much more than the single cycle used by other instructions, just how much time being determined by the current state of the cache. The original, unoptimized version of SEQUENCER's send includes 12 loads and seven stores; the optimized version has one load and two stores, all in "the real work". This reduction in the number of loads and stores is roughly proportionate to the overall reduction in the number of instructions, a factor of about six.

3.2.2 Timing Measurements

I also compared the performance of an implementation of UDP in the x-kernel with an equivalent protocol stack in Morpheus. Because UDP cannot be implemented in

Morpheus—it performs functions belonging to two different shapes—the Morpheus equivalent consists of two protocols: a multiplexor performing first-come-first-serve multiplexing, and a worker that records in the message header the length of a sent message and trims each received message to the length recorded in its header. Omission of the checksumming function is discussed below.

The purpose of this experiment was to verify whether Morpheus's purported performance advantages would result in measurably high performance. The x-kernel was used as the standard for comparison because I could obtain timing measurements for the x-kernel's UDP on the same processor (Decstation 5000/200), and because the x-kernel is known to support high performance protocol implementations [HP91]. UDP was used as the basis for comparison because, while fairly simple, it qualifies as a "real protocol," and because it has a clear Morpheus equivalent.

I measured the end-to-end latency contribution of the two versions of UDP—the time it takes UDP to send and receive one message, independent of all other protocol or hardware layers. The measurement was made by sending and receiving ten million, 1-word messages, and dividing the elapsed time by ten million. The x-kernel implementation took 24.57 microseconds, while the Morpheus equivalent took only 1.48 microseconds, a factor of 16 difference.

Two qualifications apply to this result. First, there is the issue of the accuracy of microbenchmarks and their susceptibility to cache effects. In these experiments, all messages were transmitted over the same data stream with no intervening messages, with source and destination sharing the same processor, and no flushing of the cache. This should represent a best case performance, with very little data cache effect.

Second, the figure quoted for the x-kernel is not strictly latency but also includes the time to return control through the protocol graph on both the receiving and sending sides. This returning of control would normally occur either in parallel with message transmission, or after the message has been received, but took place serially in my experiment because source and destination shared the same processor. In this particular experiment, the additional time is relatively insignificant because it only involves a total of three procedure returns. This was not a factor for the Morpheus time because Morpheus's short-circuit return optimization avoids the cost of returning for the layers being measured.

Despite these qualifications, the magnitude of the difference argues strongly for a Morpheus performance advantage. The difference is not attributable solely to Morpheus' optimizations, however; two other aspects of Morpheus also figure prominently.

First, even though UDP's checksum option was not used in the test, the x-kernel version still set the checksum field to zero on the sending side, and tested it for equality to zero on the receiving side. The Morpheus equivalent did not have this overhead. This is a legitimate advantage, attributable to building-blocks protocols approach used by Morpheus. In a protocol graph composed of many, simple protocols, the option of having a checksum is implemented by having two paths through the graph, one with the checksumming layer and one without it.

Second, accessing message headers is a far more elaborate process for the x-kernel than for Morpheus. Because compound data types such as C structures conform to alignment restrictions that may not be satisfied by the space allocated to a message header, x-kernel protocols read and write from temporary headers that are copied to and from messages by protocol-specific functions that account for potential alignment differences. Byte swapping, if necessary, is performed at the same time. Header manipulations in Morpheus are more efficient for two reasons. First, Morpheus ensures that header fields in messages satisfy its alignment restrictions. This is accomplished by padding a header internally so that individual fields are aligned with respect to the start of the header, and padding a header externally to maintain the invariant that each header starts on a word boundary. Second, any byte-swapping is performed by in-line code generated by the compiler for assignments that appear in the source language program. Hence, no function calls are required for either alignment or byte order; message headers may be read and written directly as if they were ordinary records, with any necessary byte swapping taking place invisibly and efficiently.

3.3 Discussion

Morpheus's dedicated message registers, short-circuit return, and procedure cloning optimizations are interprocedural in nature, but cannot be duplicated by interprocedural optimization of a general purpose language. Since it is not determined until runtime which protocol will be layered on top of which other protocol, it is unknown at compile time which callee procedure corresponds to a call site. Even if these optimizations could be duplicated using general interprocedural optimization, it would involve considerable interprocedural analysis at compile time. Furthermore, if separate compilation were to be supported, there would be additional compile time overhead to keep track of interprocedural dependences between separately compiled modules. Morpheus's latency optimizations, which supports separate compilation, avoid these compile time penalties. In effect, the interprocedural analysis took place at language design time.

Suppose instead that the protocol configuration were fixed at compile time, as protocol integration assumes. It would be possible to inline **sendLatency**s or **deliverLatency**s from a series of layers into a single function. A function call interface between layers would be needed only at interfaces where the sequence of layers is not fixed, e.g. demultiplexing from a multiplexor to any one of a number of higher level protocols. The dedicated message registers and short-circuit return optimizations would apply only at these function call interfaces. Interprocedural analysis could conceivably arrive at similar optimization of these function call interfaces, but would require compile time analysis. Morpheus's procedure cloning optimization would still not be duplicatable by interprocedural analysis, since the Sessions on which the clones are based are created at runtime.

CHAPTER 4
THROUGHPUT OPTIMIZATION

This chapter presents Morpheus's sole throughput optimization. Combining this optimization with Morpheus's latency optimizations makes fine-grain protocol modularity practical by reducing the performance penalty for protocol layers.

Morpheus's throughput optimization is applied only to **sendThruput** and **deliverThruput**, not the latency-optimized operations **sendLatency** and **deliverLatency**. The conclusion of this chapter discusses the possible alternatives to having distinct operations optimized for latency versus throughput.

Morpheus's throughput optimization is a compile time optimization that utilizes protocol configuration information, so it requires that the protocol configuration be bound at compile time. This contrasts with Morpheus's latency optimizations, all of which support runtime configuration. One way to obtain most of the benefits of both runtime configuration and optimizing throughput would be to configure at compile time a core protocol graph which could be optimized for throughput, and configure any additional protocols needed at runtime without the benefit of the throughput optimization.

4.1 Integrated Layer Processing

Data manipulation—e.g., encryption, presentation formatting, compression, computing checksums—is one of the costliest aspects of data transfer [CJRS89, CT90, DAPP93]. This is because reading, and possibly writing, each byte of data in a message involves memory loads or stores, which are relatively slow operations on modern RISC architectures. Load and store operations typically ranged from 8 to 32 clock cycles per memory access in 1990 [HP90], in contrast to other operations that complete in a single cycle on modern RISC architectures. Furthermore, the discrepancy between processor and memory performance is expected to get worse.

Caches offer only a partial solution to this problem. While caches are very effective in reducing memory accesses for many computations, the characteristics of strictly layered message processing are such that caching is not as effective [DAPP93]. Furthermore, there is still a cost for accessing the cache—typically 1 to 4 clock cycles in 1990 [HP90].

This cost must be paid by every data manipulation protocol, for every word of data. In addition, there are delay slots following each read access which may not all be fillable.

Clark and Tennenhouse [CT90] suggest a strategy called *Integrated Layer Processing* (ILP) for optimizing data manipulation. In this dissertation, I refer to ILP as *protocol integration*, or simply *integration*. Integration generalizes the compiler optimization known as *loop fusion*, as illustrated in Figure 4.1. The for-loops in Figure 4.1(a) model a strictly layered, serial implementation of two data manipulations, and the for-loop in Figure 4.1(b) models an integrated implementation of the same two data manipulations.

```
for( i = 0; i < 10000; i++ )
    msgData[i]++;                  /* LOAD, ADD, STORE */

for( i = 0; i < 10000; i++ )
    msgData[i] = ~msgData[i];      /* LOAD, COMPLEMENT, STORE */
```

(a) Two For−Loops

```
for( i = 0; i < 10000; i++ ){
    temp = msgData[i];             /* LOAD */
    temp++;                        /* ADD */
    temp = ~temp;                  /* COMPLEMENT */
    msgData[i] = temp;             /* STORE */
}
```

(b) Integrated For−Loops

Figure 4.1: For-Loops

When the C code in the examples is compiled to run on a RISC architecture, the data manipulation steps result in the machine instructions noted in the comments (assuming the variable *temp* is implemented as a register). In the serial for-loops, each time a word of data is manipulated, it is loaded and stored. In the integrated for loop, in contrast, each word is loaded and stored only once, even though it is manipulated twice. This is possible because the data word remains in a register between the two data manipulations. Hence, integrating the for-loops results in the elimination of one load and one store per word of data.

Abstractly, this situation can be described as follows. Memory hierarchies are optimized for locality of reference. Integration restructures a computation with poor temporal locality of data reference into one with good locality. The compiler takes advantage of this increased locality by leaving the data word in a register between manipulations.

Clark and Tennenhouse [CT90] quantify the potential advantage of this technique by fusing some simple data manipulation loops. They report a 48% improvement in throughput when combining checksum and copy, and a 7% improvement when combining ASN.1 integer conversion and checksum. These results must be qualified by noting that first, the measurements represent isolated data manipulations and not complete protocols; second, they assume, unrealistically, that no data is cached between manipulations in the serial case; and third, they measured unrolled loops.

4.1.1 Four ILP Problems

As described, ILP is more an implementation *strategy* than an applicable *technique*. Applying ILP to a protocol suite involves solving a number of implementation problems. I have identified four basic problems, although a particular protocol suite need not present all four. The problems are:

Accommodating awkward data manipulations. Some protocol data manipulations may not fit the for-loop model. Different manipulations may require different sized units of data, and some can change the quantity of data.

Reconciling different views of data. A single message looks different at different layers in a series of protocols, as layers add or remove headers. Hence, adjacent protocols do not share a common definition of what data to manipulate—one protocol's data is another protocol's header, and is nonexistent to a third protocol.

Satisfying ordering constraints. Protocol processing includes tasks other than data manipulations. These include reading and writing headers, updating connection state, and sending control messages. There are constraints on the ordering of these tasks relative to data manipulation that rule out simply extracting the data manipulations and integrating them.

Preserving modularity. Mixing code from different protocol layers compromises the modularity of protocol implementations. This makes it harder to design, implement, modify, maintain, debug, and reuse protocol implementations.

4.1.2 Related Work

The obvious approach to applying ILP is to customize an implementation for each particular suite of protocols, perhaps with different implementations for different types of machine. Solutions to the four ILP problems can be based on the characteristics of the particular protocols and machine, and, in particular, protocol modularity can be sacrificed. Before this research, the four general ILP problems had not been identified as such because researchers thought in terms of the particular form taken by those problems in the context of particular protocol suites.

In several working TCP/IP implementations, checksumming and copying have been integrated [CJRS89]. This is a degenerate case of ILP in that the two data manipulations belong to the same protocol. Hence the problems of reconciling different views of messages, satisfying ordering constraints, and preserving modularity do not arise. Furthermore, the particular data manipulations involved are regular enough to allow them to be combined in a simple for-loop.

Gunningberg, et al. [GPSV91] investigated integrating some more interesting data manipulations, and incorporated message header writing. The three data manipulations they considered were a simple presentation encoding, checksumming, and DES (Data Encryption Standard) encryption [Tan88]. These data manipulations offer some complications: the presentation encoding increases the quantity of data by inserting bytes, and DES inherently processes eight byte units. In the integrated form, the first layer (the presentation encoding) reads from a message buffer until it can output eight bytes; the data is subsequently passed between layers eight bytes at a time, in a pair of registers. A message header was generated by a technique customized to these particular data manipulations. The integrated version was compared with a strictly layered version in which the cache was flushed between data manipulations. The integrated version gave only a 0.5% increase in bandwidth over the strictly layered version on a Sun SPARC station. The authors reason that the relative improvement was small because the DES algorithm is so slow (less than 1% of the bandwidth of each of the other two data manipulations) that it dominated the overall bandwidth.[1]

The performance comparisons in the Gunningberg et al. paper, as well as those reported in [CT90], assume that no message data remains cached between data manipulations in the strictly layered case. In practice however, while caches are not highly effective for

[1] Unlike DES, most data manipulations have a low processing-to-memory ratio. Furthermore, computationally intensive data manipulations such as DES can be expected to benefit more from integration as processor speeds increase relative to memory speeds.

message processing, neither are they completely ineffective. To account for a range of possible cache effectiveness, my performance experiments consider integration at both of the extremes: when all the data remains cached, and when none remains cached.

The generality of the above techniques is very limited since they are tailored to particular protocol suites. Morpheus incorporates a general ILP technique.

4.1.3 Morpheus ILP

If one's goal were to apply ILP to a particular suite of protocols on a particular type of machine, then one could use customized solutions to the four ILP problems based on the characteristics of the particular protocols and machine. In contrast, the building-blocks approach supported by Morpheus requires very general solutions to these problems. If protocol implementations are to be reusable in different contexts, the solutions to the four ILP problems must work in the different contexts. Morpheus's solutions, briefly described, are as follows:

Accommodating inconvenient data manipulations. Each data manipulation is expressed as a function called a *word filter* which manipulates a single machine word. Word filters use state variables and control constructs to accommodate non-word units and changes in the quantity of data. For performance reasons, word filters are not implemented as functions at the object code level, but are instead combined in a single function.

Reconciling different views of data. Morpheus integrates the manipulation of just that data that all the layers agree is data—application data. Protocol layers can identify the application data portion of a message because it is exposed by a new abstract data type for messages called a *segregated* message.

Satisfying ordering constraints. Each message processing operation is executed in three stages: an *initial stage*, a *data manipulation stage*, and a *final stage*. The initial stages of a series of layers are executed in sequence, then the integrated data manipulations take place in one shared stage, and then the final stages are executed in sequence. The ordering constraints are satisfied by executing the various message processing tasks in the appropriate stages.

Preserving modularity. Morpheus preserves modularity by automating the integration process. Morpheus protocol programs are independent of each other, but combined by the compiler into a single object code level implementation. This allows protocols to be designed, implemented, modified, and maintained independently of each other. Protocols may be reused by configuring them in different combinations, and a protocol may be debugged by integrating it "by itself."

The generality of this integration technique involves a trade-off with performance: for a given protocol suite and machine, it is probably possible to customize ILP for that suite and machine in such a way as to outperform the technique presented in this chapter. This is the same trade-off that exists between programming in a high-level language and programming in assembler language: the code generated by a compiler is generally not as efficient as the code that could be directly programmed in assembler language, but the small performance loss is more than offset by the advantages of high-level languages.

Part of the Morpheus integration technique involves a more complicated representation of the internal **sendThruput** and **deliverThruput** operations than the one described in Chapter 2 for **send** and **deliver**. Instead each is expressed as a set of functions with some shared data structures. The individual functions are presented as they arise in the presentation of the Morpheus integration technique, and summarized after they have all been introduced.

4.2 Accommodating Awkward Data Manipulations

The example in the introduction to this chapter models data manipulations as for-loops, combining two for-loops into a single integrated for-loop. This is possible because the two artificial data manipulations involved both operate on the same sized unit of data, and both process data in a one-in-one-out fashion.

Data manipulations are not always so regular. Data manipulations may process different units. DES, for example, processes 64 bits at a time—it is not defined for any smaller quantity. On the other hand, TCP checksumming is based on 16-bit units.

A thornier problem is that data manipulations such as data compression or presentation formatting can change the total quantity of data. More generally, a data manipulation could produce data at a rate different from that at which it consumes it, even if the total quantity of data remains constant. This rules out the for-loop approach.

The requirement to support differing consumption and production rates is suggestive of Unix pipes, but this is misleading. The primary purpose of pipes is to decouple the rate of production at one end of a pipe from the rate of consumption at the other end. This is the opposite of what is wanted for integrated data manipulations. In terms of pipes, I want to use a very small pipe buffer—just one word—so that the buffer can be implemented as a register. Moreover, the output rate of one data manipulation must be coupled to the input rate of the next to avoid any synchronization cost.

4.2.1 Word Filters

My solution involves expressing each data manipulation as a function called a *word filter* that processes a single machine word of data, commonly 32 bits, each time it is invoked. (For performance reasons, word filters are not implemented as conventional functions; their implementation is described below.) A word filter is invoked repeatedly to process the data in a message one word at a time. In the common case, a word filter outputs one word each time a word is input, but it could instead output zero or multiple words. "Outputting" a word consists of invoking the next data manipulation's word filter with that output word as its input. Figure 4.2 shows a word filter for computing a checksum.

```
filterData(dataWord)
{
	sum += (dataWord & 0x0000FFFF) + (dataWord >> 16);
	output( dataWord );
}
```

Figure 4.2: Checksum Word Filter

Word filters accommodate data unit discrepancies and data rate changes by using control constructs and state variables. This is illustrated in Figure 4.3 using a data manipulation called PES as an example. PES (for Pseudo Encryption Standard) is an artificial data manipulation based on DES. DES exhibits interesting data manipulation characteristics but is so slow that, unlike other data manipulations, the time to perform the data manipulation itself dominates the data access time. PES is my vehicle for investigating the data access characteristics of DES without the extensive computation. PES replaces DES's extensive computation with a simple transformation of the data.

PES is like DES in that it must have 64 bits—two 32-bit machine words—at a time to perform its manipulation. The PES word filter uses a state variable as a flag to indicate whether the next word will be the first or the second of a pair. It uses a control construct, an if-statement, to vary its behavior based on the flag. When it is invoked with a first word, it does not output any words, but instead saves the input word in a state variable, and toggles the flag so that it will recognize the next input word as the second of a pair. When it is invoked with a second word, it encrypts that word together with the first word, outputs the two resulting encrypted words, and toggles the flag so that it will recognize the next input word as the first of a pair.

```
filterData(dataWord)
{
	if( ! awaitingSecondWordOfPair ){
		/* dataWord IS THE FIRST WORD OF A PAIR */
		firstWord = dataWord;
		/* DON'T OUTPUT ANYTHING */
		awaitingSecondWordOfPair = TRUE;
	}else{
		/* dataWord IS THE SECOND WORD OF A PAIR */
		output( (firstWord & 0xF0F0F0F0) | (dataWord & 0x0F0F0F0F) );
		output( (firstWord & 0x0F0F0F0F) | (dataWord & 0xF0F0F0F0) );
		awaitingSecondWordOfPair = FALSE;
	}
}
```

Figure 4.3: PES Word Filter

The use of internal state by a word filter has the consequence that a filter may end up with some output implicit in its state when there is no more input. For example, if the PES word filter is given an odd number of data words, it will not produce any output corresponding to its last input word. To handle this, such data manipulations must also have a *flush* function that outputs any output left implicit in state variables. This flush function is invoked when there is no more input data. Figure 4.4 shows how this might be implemented for PES.

```
flush()
{
	if( awaitingSecondWordOfPair ){
		/* EXPECTED A SECOND WORD WE NEVER GOT; USE A BOGUS VALUE */
		output( (firstWord & 0xF0F0F0F0) | (0x12345678 & 0x0F0F0F0F) );
		output( (firstWord & 0x0F0F0F0F) | (0x12345678 & 0xF0F0F0F0) );
	}
}
```

Figure 4.4: PES Flush

Each word filter invokes the next filter when it has a word to output. The first filter gets its input from a loop that reads the data from a message data structure, invoking the filter once per word. The last filter in the series outputs its data to a routine that looks like a filter, but simply writes its input into an output message data structure.

Note that although different units of data are optimal for different data manipulations, the word filter approach compromises by imposing a single fixed unit—the machine word—for passing data between layers. This has the advantage of simplifying the interface between protocols and avoids any runtime interpretation entailed by passing variable units. The resulting common interface for data manipulations makes it straightforward to automatically combine code from different protocols, as explained later in this chapter. The machine word is the obvious choice for the fixed unit, both because machine architectures are optimized for words, and because most data manipulations can efficiently process words. By processing a word at a time, a data manipulation whose natural unit is a byte or half-word effectively processes multiple units in parallel.

This design imposes the specification-level constraint that the data being manipulated always consists of an integral number of words. One alternative design would accommodate fractions of a word by adding "byte filters" that would be invoked on the odd bytes at the end of the data. This alternative is analogous to **bcopy** implementations.

Host machines on a network may unfortunately have different word sizes. In this case, each host would still manipulate data in units of its own native word size, but message data would be constrained to consist of an integral number of units whose size is the least common multiple of the host word sizes. If this least common multiple were excessively large (e.g. if word sizes were not all powers of two), then the alternative design using byte filters should be used.

4.2.2 Word Filter Implementation

Logically, word filters are functions, and are expressed as functions in the Morpheus source code, but implementing them as functions at the object code level would have two performance problems. The first is function call overhead: a function call is too high a price to pay each time a word of data is passed from one data manipulation to the next.

The second problem has to do with implementing word filter state variables. In general, a filter's state must persist across the invocations corresponding to a particular message. For example, a checksum's partial sum must be accumulated across invocations of the checksum filter until an entire message has been processed. Variables that persist across invocations of a function—globals, or statics—are invariably implemented in memory instead of registers. Implementing such state variables as registers would make word filters more efficient.

Morpheus solves these problems by merging adjacent word filters into a single object code level function. This is like inlining, except that inlining would not implement static local variables as registers. This avoids function call overhead, and it permits the implementation of state variables as registers since those variables are now all local to a single object code level function.

Filters are merged into a single function at the object code level, not the source code level, but I illustrate the effect of merging by presenting an analogous source code level merging in Figure 4.5. This example uses three data manipulations: BSWAP, PES, and CKSUM. BSWAP reverses byte ordering, and CKSUM computes a checksum. Note that the combined filters would be embedded in a for-loop or other iterative construct.

```
/* READ A WORD OF INPUT */
DataWord = *inputBuffer++;

    /* BSWAP */
    DataWord = ((DataWord & 0x00FF00FF) << 8) | ((DataWord & 0xFF00FF00) >> 8);

        /* PES */
        if( ! awaitingSecondWord ){     /* DataWord is first word of a pair */
                firstWord = DataWord;
                awaintingSecondWord = TRUE;

        }else{   /* DataWord is second word of a pair */
                secondWord = DataWord;
                awaitingSecondWord = FALSE;
                DataWord = (firstWord & 0xF0F0F0F0) | (secondWord & 0x0F0F0F0F);

                /* CKSUM */
                sum += (DataWord & 0xFFFF) + (DataWord >> 16);

                    /* WRITE A WORD OF OUTPUT */
                    *outputBuffer++ = DataWord;

                DataWord = (firstWord & 0x0F0F0F0F) | (secondWord & 0xF0F0F0F0);

                /* CKSUM */
                sum += (DataWord & 0xFFFF) + (DataWord >> 16);

                    /* WRITE A WORD OF OUTPUT */
                    *outputBuffer++ = DataWord;
        }
```

Figure 4.5: Combined Filters

The common case for data manipulations is the one-word-in, one-word-out behavior required by a for-loop. As more of the data manipulations in a series fit this common case, the implementation of the integrated word filters comes to more closely approximate a basic for-loop. The integrated word filter implementation is in some sense a generalized for-loop. It has the performance of a for-loop in the common case, but is flexible enough to accommodate more awkward data manipulations.

4.3 Measurements and Analysis

This section reports the results of experiments comparing serial data manipulations to data manipulations integrated using word filters, and investigates the limits of integration. These experiments involve only isolated data manipulations; experiments involving complete protocols are reported later in this chapter.

The following experiments are based on data manipulations written in C. To simulate a Morpheus compiler's merging of word filters at the object code level, the integrated implementations consist of word filters merged into a single function at the C source code level. A data word is passed from one filter to the next at the C level by leaving it in a variable that is shared by the filters, and the C compiler implements the variable as a register. It also implements the word filter state variables, which are local to the combined function, as registers. Hence the object code generated for integrated data manipulations by a C compiler gives a close approximation to the object code that would be generated by a Morpheus compiler.

Data manipulation performance depends heavily on cache effects, i.e. whether or not the loads and stores actually result in memory accesses. My experiments accounted for this by measuring performance at each of the two extremes: when all the message data is in the cache, and when none is in the cache. Thus, the actual performance in any particular situation must be within the range specified by the pair of measurements. To obtain cache hits, the data was accessed before beginning data manipulations, and the data manipulations were run back-to-back with no concurrent processing, using buffers sized and situated so as to avoid collisions in the cache. Note that in this cache hit case, even the first data manipulation of a series found all the data in cache. To obtain cache misses the pertinent part of the cache was flushed between data manipulations, and the time to flush was subtracted out. Cache behavior also depends on the particular data manipulations, so I have measured several combinations of data manipulations.

4.3.1 Experimental Platforms

Cache behavior also depends on the particular machine architecture. Each experiment was conducted on three different machines—the DecStation 5000/200, the HP 720, and the SPARCstation 1—to verify that the word filter technique is not specific to a particular machine or cache design. On all three machines, the experiments used the native C compiler with the highest level of optimization.

The MIPS R3000-based DecStation 5000/200 has separate 64-KByte instruction and data caches. The caches are direct mapped using physical addresses, and cache lines are 16 bytes. Loading a word from cache costs one cycle, and entails a one cycle delay slot. Storing a word normally takes a single cycle since the data cache is write-through, with a write buffer that can buffer stores for up to six pending writes and retire writes within the same page once per cycle. It takes 13 cycles to load a cache line from memory, but the processor can access the first word after ten cycles.

The HP 720 has a 128-KByte instruction cache, and a 256-KByte copyback data cache. The caches are direct mapped and virtually addressed, and cache lines are 32 bytes. Filling a cache line takes 18 cycles if the replaced line is clean, 23 cycles if it is dirty. Storing to an uncached location takes 23 cycles if the replaced line is clean, 27 cycles if it is dirty. Loads or stores of cached words execute in a single cycle.

The SPARCstation 1 has a 64-KByte combined write-through cache. It is virtually addressed, with a line size of 16 bytes. If the referenced line is in cache, loading a single word takes two cycles, and storing a single word takes three cycles. Several stores in succession will cause a stall once the memory write buffers fill, even if the target addresses are all cached. If the referenced word is not in cache, a single word load takes 14 cycles, and a single word store takes six cycles.

4.3.2 Case Study

I timed three combinations of CKSUM, BSWAP, and PES. These data manipulations were chosen because they are representative of actual protocols, and they provide examples of both read-only and read-write data manipulations, and of both simple and convoluted filters.

Tables 4.1 and 4.2 give the results when the data is and is not cached, respectively. These tables show, for example, that on the DecStation 5000, integrating CKSUM and BSWAP increases bandwidth from 44.7 Mbps to 54.4 Mbps if it is cached, and from 29.3 Mbps to 39.6 Mbps if all message data is uncached.

	DS5000		HP720		Sparc1	
	Serial (Mbps)	Integrated (Mbps)	Serial (Mbps)	Integrated (Mbps)	Serial (Mbps)	Integrated (Mbps)
CKSUM+BSWAP	44.7	54.4	84.2	101.0	24.0	30.0
BSWAP+PES	33.9	43.5	72.1	101.0	17.1	25.0
BSWAP+PES+CKSUM	25.8	35.4	54.1	79.7	13.6	21.1

Table 4.1: Serial vs Integrated When Data in Cache.

The throughput improvements resulting from integration are summarized in Figure 4.3. The table shows, for example, that the integrated implementation of CKSUM and BSWAP

	DS5000		HP720		Sparc1	
	Serial (Mbps)	Integrated (Mbps)	Serial (Mbps)	Integrated (Mbps)	Serial (Mbps)	Integrated (Mbps)
CKSUM+BSWAP	29.3	39.6	42.4	55.8	19.3	26.1
BSWAP+PES	23.4	34.0	33.1	56.2	14.6	22.2
BSWAP+PES+CKSUM	17.5	29.0	26.2	48.9	11.3	19.0

Table 4.2: Serial vs Integrated When Data Not in Cache.

on the DecStation has 22% to 35% higher bandwidth than the serial implementation. The low end of each range (22% in this example) applies when all of the message data is cached, while the high end applies when none is cached. The improvement achieved in any particular situation would necessarily fall between these two numbers.

	DS5000 (percent)	HP720 (percent)	Sparc1 (percent)
CKSUM+BSWAP	22 – 35	20 – 32	25 – 35
BSWAP+PES	28 – 45	40 – 70	46 – 52
BSWAP+PES+CKSUM	37 – 66	47 – 87	55 – 68

Table 4.3: Bandwidth Improvement due to Integration.

Several factors contribute to integration improving throughput. Most obviously, integration eliminates the load and store instructions themselves. This not only saves the time to transfer data between memory and the cache, but also saves the time to load the data from the cache and store data to the cache or a write buffer. Another major factor is the reduced loop overhead, since integrated data manipulations share a single loop through the data instead of iterating through the message data once for each data manipulation. Since there are fewer load and store instructions and they occur in an inner loop containing more instructions, the delay slots after loads are more likely to be filled, and on machines with write-through caches and write buffers, stores are less likely to fill a write buffer. Finally, integration eliminates some buffer allocations and deallocations since data which would have been buffered between data manipulations is now streamed directly from one to the next.

To verify that the most important factors have been accounted for, I compared the time saved by integrating to the time taken by loops that simply load and/or store one word of data per iteration. Table 4.4 presents the results for the DecStation. The measurements are given in units of seconds per MByte. These particular combinations of data manipulations give one example each of eliminating a load, a load and a store, and two loads and a store. The table shows, for example, that integrating CKSUM and BSWAP reduces the time to process one MByte of data by about 0.071 seconds if none of the data is cached, which is roughly equal to the 0.082 seconds it takes to load one MByte of data from main memory. Context-dependent factors, such as unfilled delay slots, could account for the small discrepancy.

	Cache Hit	
	Improvement (sec/MB)	Comparison (sec/MB)
CKSUM+BSWAP	.032	Load=.042
BSWAP+PES	.052	Load+Store=.053
BSWAP+PES+CKSUM	.084	2×Load+Store=.095
	Cache Miss	
	Improvement (sec/MB)	Comparison (sec/MB)
CKSUM+BSWAP	.071	Load=.082
BSWAP+PES	.107	Load+Store=.097
BSWAP+PES+CKSUM	.180	2×Load+Store=.179

Table 4.4: Comparison of Integration Savings.

4.3.3 Scalability

The preceding experiments show the effects of integrating two or three data manipulations. When greater numbers of layers are integrated, register pressure becomes a factor because combining multiple data manipulations in a single function increases the demand for the registers for heavily accessed variables and constants. Consequently, integrating many layers may spill registers more than serial implementations where each data manipulation has the entire register set to itself.

I investigated how integration scales by combining varying numbers of layers, where each layer performed the identical data manipulation but used its own two local variables. The data manipulation is intended to represent a "typical" data manipulation. The results for the DecStation are presented in Figure 4.6.

The vertical dimension is in units of seconds per MByte instead of Mbps in order to expose linear behavior; Mbps would result in exponential decay curves, obscuring the knee in the integrated implementation plots. The plots are otherwise linear because each additional layer adds the same amount of time to the processing of a MByte of data. The knees in the integrated implementation plots occur at the point where the number of variables exceeds the available registers. Each layer of data manipulation uses its

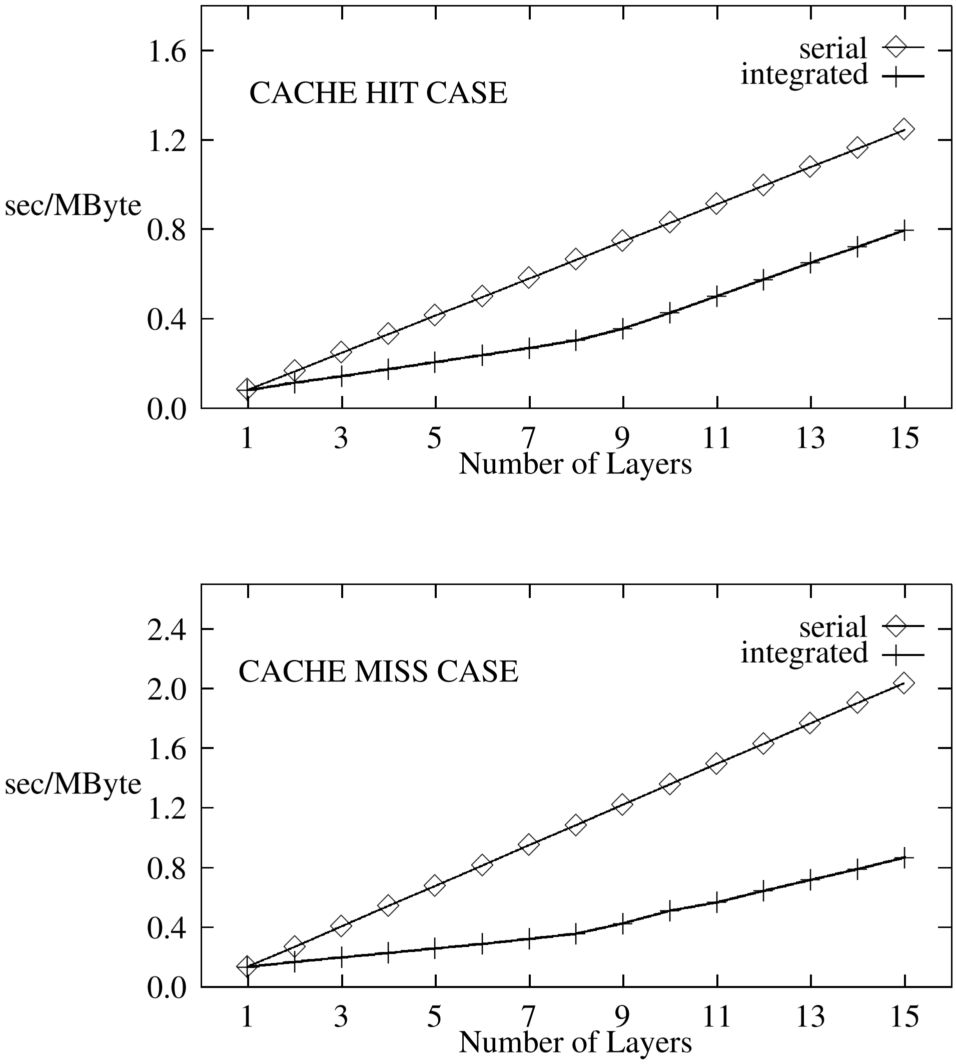


Figure 4.6: Incremental Performance On DecStation

own two local variables. When only a few layers are integrated, these variables are all implemented as registers. As additional layers are integrated, the additional variables must be implemented on the stack. Hence, the cost of an additional layer is smaller when there are still registers available for variables, and greater when there are more variables than usable registers.

At the knee in the DecStation case (eight layers), the integrated version runs 2.2 times faster than the serial in the cached case, and 3.0 times faster in the uncached case. Table 4.5 reports where the knee occurs and the associated improvements factors for the three machines I tested. Notice that the difference in the improvement factor in the cached and uncached case becomes more dramatic as the processor becomes faster—in this case, the HP720 is faster than the DS5000, which is faster than the Sparc1.

	DS5000	HP720	Sparc1
Knee (number of layers)	8	7	5
Improvement ($\times$ faster)	2.2 – 3.0	2.0 – 3.8	2.6 – 2.8

Table 4.5: Improvement Factor at Knee of the Integration Curve.

The location of the knee, and the slope beyond the knee, both depend on the ratio of local variables to layers. Given a high ratio and large number of layers, an integrated implementation could perform worse than a serial implementation. However, one would not integrate this many layers all in one series; instead one would break off integration whenever the knee is reached, and start a new integrated series, resulting in smaller integrated series that do not exhaust the register set. This results in performance that is invariably better that serial implementations. Nonetheless, a larger register set would support even better performance by allowing more layers to be integrated before hitting the knee. As the discrepancy between processor and memory performance widens, even applications may recast an increasing number of their data transformations into a form that can be integrated.

4.3.4 Performance Prediction Model

Having reported the performance of integration on existing machines, I now present a model for predicting its performance on future machines. The potential performance of integration on future machines will depend not only on processor and memory speeds, but also on the sizes, speeds, and number of caches. Processor and memory speeds of the near future are reasonably predictable because they tend to improve at fairly constant rates, but caches are more difficult to predict, in part because they change in response to changes in the ratio of processor speed to memory speed. Any model predicated on a particular cache design would be highly questionable. Therefore the model presented here does not predict cache performance, but instead captures it in a variable parameter, allowing the reader to make performance projections based on his or her own best guess about future hardware.

The model counts the cycles taken by the three components of a data manipulation:

Loop overhead. Loop overhead is the incrementing and branching instructions that make a loop. The model assumes that a loop takes three cycles independent of the particular machine: one to increment the pointer to the location to load from, one to increment the store pointer (assuming for simplicity that each data manipulation is read-write), and one to branch. Thus, an integrated implementation uses 3 cycles per data word for loop overhead. Since a serial implementation pays this loop overhead at each layer, it uses $3 \times num\_layers$ cycles per data word, where num_layers is the number of data manipulation layers.

Computation. Computation is the manipulation of the data while it is in registers. The model assumes that the number of cycles for the computation part of a particular data manipulation remains constant independently of the particular machine. It also assumes that local variables are implemented as registers, so the only data being loaded or stored is the data being manipulated. The cycles spent on computation are the same in both the integrated and serial cases. The number of cycles spent on computation is represented in the model by the parameter $computation$.

Data access. Data access is the load and store instructions. The average number of cycles taken by these is captured in a parameter called $data\_access\_cycles$. For simplicity, loads and stores are modeled as taking the same number of cycles on average. Thus, $data\_access\_cycles$ combines memory and cache speeds and cache hit/miss ratios into one parameter. For example, on a machine that takes ten cycles to go to memory and two cycles to go to cache, if one were to assume a 50% cache hit rate, then one would use (0.5 $\times$ 10) + (0.5 $\times$ 2) = 6 as the estimated $data\_access\_cycles$.

The number of data access cycles per data word for a serial implementation is

$$data\_access\_cycles \times (num\_layers + num\_writers),$$

where $num\_writers$ is the number of layers that modify the data (all layers read the data, but not all layers modify the data since some are read-only). Since an integrated implementation only loads the data once and stores it once regardless of the number of layers, it uses $data\_access\_cycles \times 2$ cycles per word.

	Loop Overhead	+Computation	+Data Access
Serial	$3 \times num\_layers$	$+computation$	$+data\_access\_cycles \times (num\_layers + num\_writers)$
Integrated	3	$+computation$	$+data\_access\_cycles \times 2$

Table 4.6: Estimated Cycles to Manipulate One Data Word

Thus, the model estimates the number of cycles per data word according to the formulas in Table 4.6.

The relative increase in throughput due to integration is

$$serial\_cycles/integrated\_cycles - 1.$$

Figure 4.7 plots relative increase as a function of $data\_access\_cycles$ for the three combinations of data manipulations measured earlier.

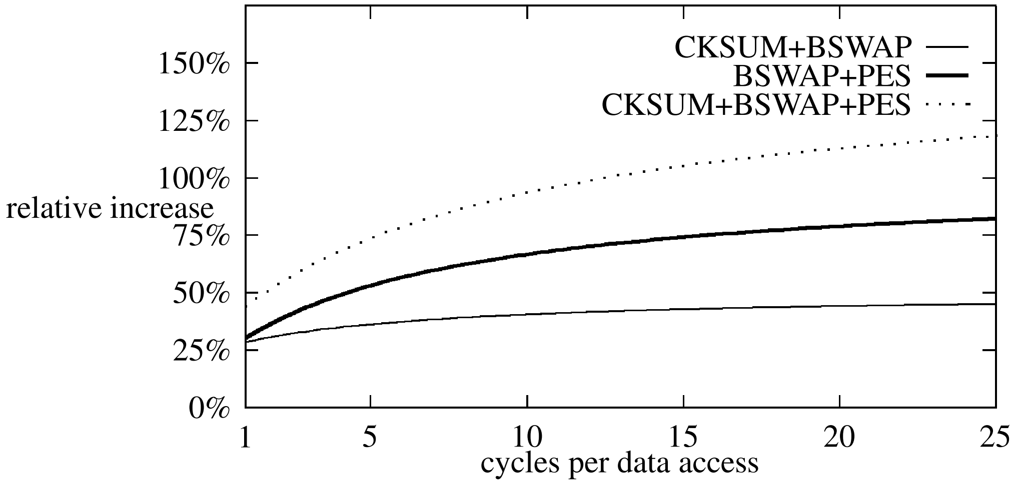


Figure 4.7: Relative Increase in Throughput due to Integration

Notice that as $data\_access\_cycles$ increases, the relative increase asymptotically approaches $(num\_layers + num\_writers)/2 - 1$. Thus, for example, integrating CKSUM and BSWAP involves two layers ($num\_layers = 2$), and one of the layers writes the data ($num\_writers = 1$), so it can achieve no better than a (2+1)/2 - 1 = 50% improvement in throughput, regardless of the number of cycles it takes to go to memory.

4.3.5 Code Space

Combining filters into a single object code level function raises the concern of using excessive space. The object code space could potentially be exponential in the number of integrated data manipulations because each filter's object code may be duplicated in more than one place in the code of the preceding filter. This is unlikely to be a problem in practice, however, since most filters are invoked in only one place in the preceding filter, the number of integrated data manipulations will be modest, and filters are relatively small bodies of code. If it were to become a problem, space consumption could be reduced by breaking a long series of integrated data manipulations into multiple shorter series.

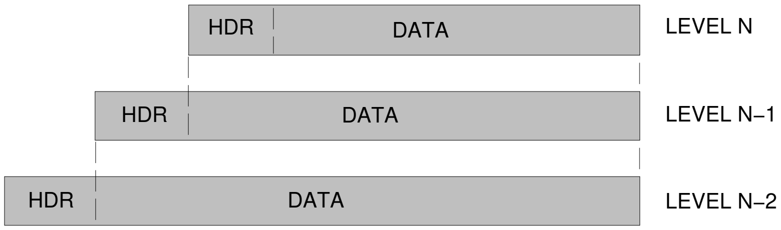
Conceivably, combining code from several data manipulations could have a negative effect on caching due to the decreased locality of reference to instructions. Integration leads to iterating through the code belonging to a whole series of data manipulations, instead of iterating through the smaller bodies of code representing an individual data manipulations. Instruction locality may be further reduced by duplicating a data manipulation inline at multiple points in the program text. To some extent, integration trades locality of instruction references for locality of data references. Yet, this larger working set of instructions is still very small compared to current cache sizes such as the DecStation 5000/200's 64-KByte instruction cache, the HP 720's 128-KByte instruction cache, and the Sparc1's 64-KByte combined cache.

I have not observed any degradation attributable to combining filters into a single function in experiments integrating up to 15 (simple) layers. Combining filters in a function is similar to inlining, and in general, inlining seems to have little effect on caching and virtual memory. Experiments reported in [CHT91] showed no obvious evidence of either instruction cache overflow or thrashing, and the previous reports they cited showed similar results.

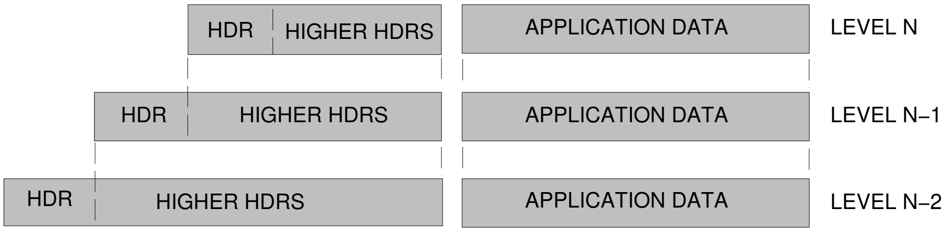
4.4 Reconciling Different Views of Data

The preceding section addressed the problem of integrating arbitrary data manipulations in isolation, outside the context of protocols. This section addresses a problem that emerges when the data to be manipulated is part of a message.

When data manipulations are integrated in the context of a protocol, the data is part of a message, not just an array. Messages change as they pass through different protocol layers. When a protocol sends a message, it generally prepends a header which its receiving peer knows how to interpret. That receiving peer deletes the header before delivering the message to a higher level protocol. Lower protocol layers cannot distinguish that header from the rest of the data—it is all opaque data (Figure 4.8a). This is a form of hierarchical encapsulation. The purpose of hierarchical encapsulation is modularity, i.e., avoiding



(a) Encapsulated Message Abstraction



(b) Segregated Message Abstraction

Figure 4.8: Message Abstractions

dependencies between protocols.

Hierarchical encapsulation complicates integration. The problem is that even though adjacent layers are to collaborate in the manipulation of data, they disagree on what part of a message is the data to be manipulated. One layer's word of data may be viewed as part of a header at another layer, and at yet another layer, it may not be part of the message at all.

Consider two particular situations. First, a sending protocol may not be able to write its header until its data manipulation is complete (e.g., CKSUM). This is a problem because any lower layer integrated with that protocol must manipulate its data in lock step with the layer in question, even though that layer's headers are not available until afterward. Second, a receiving protocol may not know what to do with its data, or even what portion of the message is its data (e.g., because of variable length headers), until it reads its header. This is a problem because that protocol must manipulate its data in lock step with any lower layer with which it is integrated, even though its header is part of the data to be manipulated by the lower layer.

One solution would be for protocols to use trailers instead of headers on the sending side, so that each protocol's "header" information would be available before the subsequent layer needs to manipulate it. Receiving side protocols would then process a message "backwards," with trailers effectively becoming headers, so that receiving protocols would have their header information available before performing data manipulation.

Despite its simplicity, this approach is inefficient because, on the receiving side, a protocol would have to distinguish words of data from words of header. This entails the execution of a conditional by every layer for every input word, i.e. testing a flag that indicates whether all of the header has been processed. This approach also complicates programming, since headers must be generated and consumed serially.

Morpheus's solution is to integrate manipulation of just that data that all the layers agree is data—the application data. By application data I mean the part of the message that corresponds to data being transmitted by an application, i.e. the part that does not include any protocol headers. This allows use of the word filter approach to integrating data manipulation without adding any complication due to headers. Protocols can identify the application data portion of a message because it is exposed by a new abstract data type for messages called a *segregated* message (Figure 4.8b).

Manipulating higher level headers separately, unintegrated, does not affect performance and it simplifies programming. Messages in throughput-dominant traffic are large, with very small headers relative to the size of the application data, so the potential benefit of integrating manipulation of headers is insignificant. Header fields can be randomly accessed in the familiar, convenient manner since their manipulation is not integrated.

The disadvantage of this approach is that if higher level headers are to be manipulated, they must be manipulated separately from the integrated manipulation of application data. This could result in additional programmer effort if the same data manipulation must be coded twice, once for application data and once for higher level headers. On the other hand, decoupling the two manipulations can be advantageous. It may be more appropriate for a given protocol to manipulate either only the application data (e.g. data compression), or only the combined headers (e.g. checksum), or perhaps both but with two different manipulations.

Protocols are able to identify the application data part of a message because Morpheus represents messages using a new abstract data type called *segregated messages* (Figure 4.8b). The only difference between a segregated message and the conventional encapsulated message is that the start of the application data is visible. This represents a relaxation of strict hierarchical encapsulation, but still preserves modularity by avoiding dependencies between specific protocols because the contents of the two parts remain opaque to lower level protocols.

The implementation of the segregated message abstract data type is like that of encapsulated messages except that it adds a pointer to the start of the application data. This requires determining the start of the application data when a network device delivers a message, since the hardware represents a message as an undifferentiated sequence of bytes. Elevating that sequence of bytes into a segregated message requires rediscovering the start of the application data. Network device drivers accomplish this by adding a header that records where the application data starts.

Segregated messages involve a constraint on protocol specifications: a specification must allow application data to be manipulated independently of higher level headers. Without this constraint a protocol specification could define a data manipulation that spanned the boundary between headers and application data, requiring some bytes from each to compute the manipulation.

4.5 Satisfying Ordering Constraints

The preceding sections have shown how to integrate manipulations of message data by using segregated messages to establish a common definition of the data, and word filters to perform the actual data manipulation. However, to integrate complete **sendThruput** or **deliverThruput** operations it is not sufficient to simply extract and integrate the data manipulation code, leaving the rest of the code unchanged. For example, suppose there were a protocol that keeps track of the sequence numbers of received messages, on top of a protocol that discards messages whose checksum is not correct. The danger that arises when these protocols are integrated is that the checksum protocol will end up rejecting a corrupted message (based on a checksum computed in its data manipulation), but the sequence number protocol will update its state (independent of any data manipulation) as if it had received the message. This would result in incorrect behavior, i.e. behavior inconsistent with a serial implementation.

In general there are ordering constraints between the various tasks performed by a **sendThruput** or **deliverThruput**. Violating ordering constraints can result in incorrect behavior, as in the above example; or it could cause protocols to fail immediately, e.g., if one layer interpreted another layer's header as its own, because the previous layer had not yet removed its header. These constraints are trivially satisfied by serially executed layers; they only come into play because integration involves overlapping the execution of different protocols, a limited form of concurrency.

4.5.1 Ordering Constraints on Tasks

SendThruput and **deliverThruput** operations perform tasks other than data manipulation. These include reading and writing headers; initializing data manipulation variables; updating protocol state; setting and clearing timers; sending control messages; passing non-message information (such as flow control) to adjacent layers; demultiplexing or making routing decisions, and (assuming segregated messages) performing data manipulation on the combined higher level headers. Of course, a given protocol might not involve all of these. I combine message processing tasks into the following categories:

Data manipulation. Reading and writing application data.

Header processing. This includes reading and writing headers, *and* manipulating higher level headers.

External behavior. This includes externally visible actions such as passing messages to adjacent layers, initiating messages such as acknowledgements, and invoking non-message operations on other layers, for example to pass congestion control information. *It also includes updating protocol state* such as updating the sequence number associated with a connection to reflect that a message with the previous number has been received. Updating state is included in the external behavior category because it can influence future external behavior.

Tasks within a protocol are subject to internal ordering constraints. For example, a checksum protocol cannot write the checksum into the message header (header processing) until after it has computed a checksum on the data (data manipulation). These are the sort of ordering constraints that are so basic to a computation that they are not normally thought of as constraints.

There are also ordering constraints *between* layers. For example, an encryption protocol must decrypt higher level headers (header processing) in a received message before the next layer can read its header (header processing). Particular ordering constraints are not normally distinguished since they are all automatically satisfied by the serial execution of layers. The danger that arises when layers are integrated is that there may be conflicts between a protocol's internal constraints, and the external constraints determined by adjacent protocols.

Consider ordering constraints on header processing. On the sending side, a header may depend on the results of data manipulation (e.g., checksum), in which case it cannot be written until after data manipulation. A layer cannot perform data manipulation of higher level headers until they have been written. Assuming headers are stacked in contiguous memory, a protocol cannot know where to write its own header until the previous header has been written and possibly manipulated.

Analogous conditions hold on the receiving side. A protocol may not be able to manipulate the data in a message without having first read its header from the message. A protocol cannot read its header until the previous protocols have performed any data manipulations on the combined headers. A protocol cannot even determine where its header starts until the previous protocols have all consumed their headers if consecutive headers are stacked in contiguous memory.

Now consider ordering constraints on external behavior. These follow from the possibility of rejecting a message. A message may be rejected by a protocol for a variety of reasons; for example, it may be a duplicate, have an illegal format, or require unavailable resources. Message rejection is more common on the receiving side, but may also occur on the sending side, for example as a result of congestion control.

Message rejection is a potential problem for protocol integration because a message may be rejected "in the middle" of an integrated series of protocols. Layers which are logically subsequent to the rejecting layer may have already begun processing the message. Furthermore, layers which logically precede the rejecting layer may not have completed processing the message; in particular, it may be that in a serial implementation, a preceding layer would have rejected the message before it ever arrived at the current rejecting layer! The requirement to behave consistently with serial implementations imposes ordering constraints on the message rejection and message acceptance code—i.e. external behavior—of different layers.

4.5.2 A Task Ordering Discipline

In effect, serial implementations adhere to an unnecessarily restrictive but simple ordering discipline: **sendThruput**s or **deliverThruput**s must be executed serially. Serial order is guaranteed to satisfy all the specific lowest level ordering constraints that represent the data dependencies resulting from a layered specification, as long as each protocol is written to satisfy its own internal ordering constraints. Integration requires finding a replacement ordering discipline that similarly satisfies all data dependencies required for a layered behavior, but nonetheless permits layers to be overlapped to the extent of combining data manipulations.

Morpheus replaces serial order with a task-level ordering discipline. It is less restrictive than serial order, but more complex. However, the complexity is limited by expressing the discipline in terms of the three message processing tasks, rather than more primitive operations.

Morpheus's ordering discipline is expressed in terms of an assignment of tasks to stages in an execution model. In this execution model, **sendThruput**s and **deliverThruput**s are executed in three stages: an *initial stage*, a *data manipulation stage*, and a *final stage*. The initial stages of a series of layers are executed serially, then the integrated data manipulations take place in one shared stage, and then the final stages are executed serially. The relationship between the three stages is illustrated in Figure 4.9.

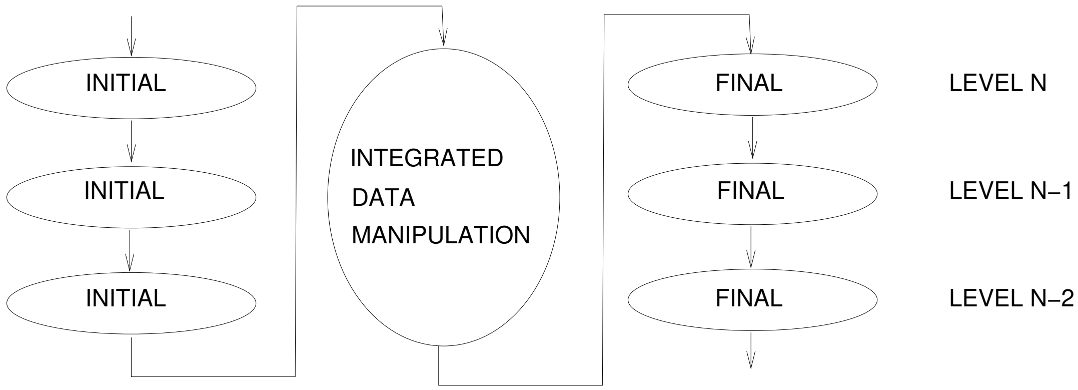


Figure 4.9: Execution Sequence of Integrated Protocol Stages

The tasks are assigned to the stages as listed in Table 4.7. Within a stage, a protocol is free to perform the tasks of that stage in any way and in any order consistent with its internal constraints. Executing the tasks in the appropriate stages ensures that the external constraints protocols impose on each other cannot conflict with their internal constraints.

TASK	STAGE
Header processing (delivering)	Initial
Data manipulation	Integrated
Header processing (sending)	Final
External behavior	Final

Table 4.7: Each Task Must Be Executed In The Corresponding Stage

An operation's initial and final stages are represented as functions. Initial and final stages for a checksum protocol's **sendThruput** are illustrated in Figures 4.10 and 4.11.

This ordering discipline resolves the message rejection problem by serializing message acceptances and rejections. Message acceptance/rejection is deferred to the final stage, which represents a sort of "commit stage." Deferring message rejection gives logically

```
initialStage()
{
	sum = msg.hdr.chksum;
	nextInitialStage();
}
```

Figure 4.10: Checksum **deliverThruput** Initial Stage

```
finalStage()
{
	while( sum & 0xFFFF0000 )
		sum = (sum & 0x0000FFFF) + (sum >> 16);
	if( sum == 0 )
		nextFinalStage();
	else
		nextAbort();
}
```

Figure 4.11: Checksum **deliverThruput** Final Stage

preceding layers the opportunity to reject the message first. The final stages execute serially, so the rejecting layer that is logically earliest will be the layer that actually ends up rejecting the message. Intuitively, the assignment of external behavior to the final stage treats the initial and data manipulation stages as if they were merely peeking at a message that is not really passed until the final stage.

Rejecting messages in an integrated implementation admits another complication. When a message is rejected by a layer, the final stages of subsequent layers are not executed. Therefore, a protocol that allocates a data structure in its initial stage, and normally deallocates it in its final stage, would not have the opportunity to deallocate. The solution is to introduce an *abort stage* that is an alternative to the final stage—if a message is rejected by a preceding layer, the abort stage is executed instead of the final stage, providing the opportunity to deallocate data structures.

4.5.3 Performance of Integrated Protocols

Earlier, this chapter reported performance measurements for isolated data manipulations—manipulations that operated on an array instead of a message, outside the context of a protocol operation. Now performance measurements are reported for complete **sendThruput**s and **deliverThruput**s, where the integrated implementations use segregated messages and the three-stage execution model. These measurements confirm that the Morpheus technique for integrating complete operations conserves most of the performance benefit of integrating isolated data manipulations.

I measured the throughput of integrated **sendThruput**s through three different combinations of protocols. The individual protocols were based on the CKSUM, BSWAP, and PES data manipulations. The measurements were made on the DecStation 5000/200, and I did not coerce the cache behavior. Table 4.8 compares serial and integrated implementations. For example, integrating the CKSUM, BSWAP, and PES *protocols* increased throughput from 18.8 Mbps to 25.6 Mbps, an improvement of about 36%. By way of comparison, integrating the corresponding three *data manipulations* gave a 37% to 66% improvement, as reported earlier.

DATA MANIPULATIONS	SERIAL (Mbps)	INTEGRATED (Mbps)	IMPROVEMENT (% of serial)
CKSUM+BSWAP	34.6	40.6	17%
BSWAP+PES	22.9	29.5	29%
BSWAP+PES+CKSUM	18.8	25.6	36%

Table 4.8: Serial vs Integrated **sendThruput**s.

These measurements represent an unrealistically high percentage of cache hits since the message size was modest (one page) and there were no concurrent processes, so the performance gap between serial and integrated implementations would normally be somewhat greater. Despite the high percentage of cache hits, the throughputs reported here tend to be somewhat low—roughly the cache miss case throughputs of the corresponding isolated data manipulations. This is due to the overhead of the protocol tasks other than the data manipulation, and the use of a different compiler, the GNU C compiler (gcc), which generates less efficient code for these particular data manipulations.

These measurements provide a rough estimate of the increase in end-to-end throughput that can be expected to result from integration. Protocol stack throughput is the primary determiner of end-to-end throughput because the throughput of high performance network hardware is higher than protocol stack throughput. The difference between end-to-end throughput and protocol stack throughput is due to synchronization overhead between a protocol stack and a network device in the form of interrupt handling and contention for the memory system. Integration should reduce this synchronization overhead as well, because it reduces contention for memory, and because interrupt handling is more expensive for serial implementations, since they have more context (i.e. cached message data) to lose.

4.6 Integration by Compiler

The solutions presented thus far—word filters, segregated messages, and the task-level ordering discipline—constitute a complete integration technique. However, direct application of this technique by a programmer would sacrifice protocol modularity by mixing code from different protocols in a single function. This would make it difficult to design, implement, modify, maintain, debug, and reuse protocol implementations.

Using Morpheus, integration is applied by the compiler rather than the programmer. This preserves modularity by adding a level of indirection to the integration technique: protocols are modular at the source code level—the programmer level—even though they are combined at the object code level—the performance level. Protocols are expressed independently of each other, and the compiler automatically combines them into integrated implementations. This situation is analogous to function inlining: an inlined function is a distinct function at the source code level, but it is compiled into code that is not a function at the object code level.

The Morpheus source code for a **sendThruput** or **deliverThruput** consists of a complete set of the components that have been introduced in the course of this chapter. The Morpheus code for a checksum protocol's **deliverThruput** is illustrated in Figure 4.12.

Corresponding components from different protocols conform to a common interface, e.g. all word filters have the same interface. In effect, the standard **sendThruput** and **deliverThruput** interfaces have been replaced by new interfaces consisting of the union of the interfaces between corresponding parts. Since the parts conform to standard interfaces, protocols can be designed, implemented, modified, and maintained independently of each other. The protocols can be freely configured in different combinations, allowing protocols to be added or deleted from protocol suites, or reused in different protocol suites. A given protocol could be configured as either integrated with other protocols, or integrated "by itself", so it can be debugged as a separate, unintegrated layer.

The standardized interfaces make it straightforward to generate integrated object code from the source code components by concatenating the object code corresponding to the components in the proper order along with the necessary infrastructure code, as depicted in Figure 4.13.

I devised a prototype integration tool that accepts protocols represented as fragments of C code and outputs integrated implementations in C. This tool was used to synthesize the integrated **sendThruput**s whose performance was reported above. Using a high-level target language isolates integration from other language implementation issues. The

```
deliverThruput{

        unsigned sum;

        initialStage()
        {
                sum = msg.hdr.chksum;
                nextInitialStage();
        }

        filterData(dataWord)
        {
                sum += (dataWord & 0x0000FFFF) + (dataWord >> 16);
                output( dataWord );
        }

        /* checksum has no flush */

        finalStage()
        {
                while( sum & 0xFFFF0000 )
                        sum = (sum & 0x0000FFFF) + (sum >> 16);
                if( sum == 0 )
                        nextFinalStage();
                else
                        nextAbort();
        }

        /* checksum has no abort */

}
```

Figure 4.12: Checksum **deliverThruput**

target language's compiler then has the responsibility for exploiting the greater locality of reference exhibited by the integrated protocols, e.g. by using registers to hold message data between data manipulations. In my experiments, the compilers have done so successfully, utilizing registers as expected.

Specifically, the prototype integrator is simply the macro processor *m4* [KR86] applied to appropriate input and include files. A protocol's **sendThruput** or **deliverThruput** operation is input as seven code fragments. To illustrate this, the CKSUM **deliverThruput** shown in Figure 4.12 is rewritten as the C code fragments in Table 4.9. The INCLUSIONS fragment is for including header files and making type or macro definitions; the DECLARATIONS fragment is for declaring local variables; and the INITIAL and FINAL fragments correspond to the initial and final stages, respectively. The ABORT fragment is used in the event of an earlier layer rejecting a message to deallocate data structures that

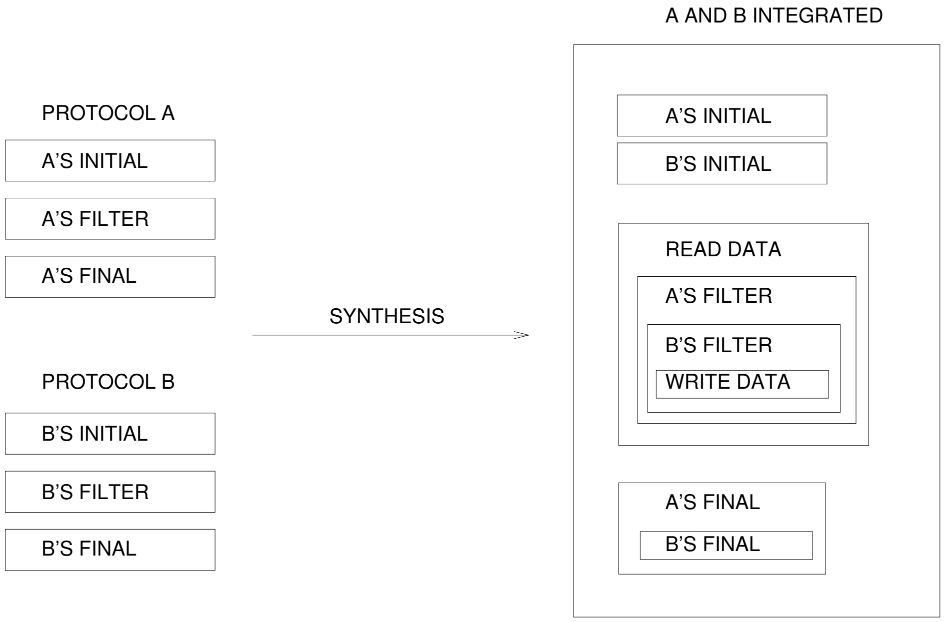


Figure 4.13: Integrated Protocol Synthesis

may have allocated by the INITIAL fragment. The two remaining fragments correspond to the data manipulation itself: FILTER is the data filter invoked by the preceding protocol when it passes a word of data, and FLUSH provides a way to flush data saved in the local state to subsequent layers once there is no more data forthcoming from preceding layers. In the case of CKSUM, there is no code to execute for a flush or abort.

The prototype integrator inserts additional code, e.g. for reading the data from the message buffer and feeding it to the first layer's word filter. This infrastructure serves as a framework in which all the protocol fragments are embedded inline, thereby establishing the sequencing of stages.

Combining programmer code verbatim, as the prototype does, has several shortcomings. The source syntax is unintuitive and inflexible. The programmer must be careful to use names that do not clash with those used in other protocols. This includes not only variable names, but also names of types, defined constants, macros, functions, and labels. The brute-force strategy I used in prototyping was to assign a globally unique name to each protocol (e.g. "cs_" for CKSUM), and prefix that name to every name used by the protocol. Another problem is that it is difficult to determine which protocol is responsible for a runtime error because there is no way to associate an object code instruction with the protocol from which it is derived.

Morpheus avoids these problems by incorporating integration in its design. The parts

FRAGMENT	CHECKSUM EXAMPLE
INCLUSIONS	#include "chksum.h"
DECLARATIONS	cs_Session cs_session; cs_Hdr cs_hdr; register unsigned cs_sum;

INITIAL	cs_session = (cs_Session) next_session;
	next_session = (GenericSession) cs_session->nextSession;
	cs_hdr = (cs_Hdr) msgPush(hdrMsgP, cs_HDR_SZ);
	cs_sum = cs_hdr->chksum;
	include(CS_NEXT/initialStage)

FILTER	cs_sum += (currWd & 0x0000FFFF) + (currWd >> 16); include(CS_NEXT/filter)
FLUSH	/* checksum has no flush */

FINAL	while(cs_sum & 0xFFFF0000)
	    cs_sum = (cs_sum & 0x0000FFFF) + (cs_sum >> 16);
	if(cs_sum == 0)
	    include(CS_NEXT/finalStage)
	else
	    include(CS_NEXT/abort)

ABORT	/* checksum has no abort */

Table 4.9: CKSUM **deliverThruput** C Code Fragments

of **sendThruput** and **deliverThruput** are expressed using intuitive, familiar constructs such as functions, with name scope rules that eliminate the danger of name clashes. A compiler could save information for attributing a runtime error to the particular offending protocol.

Note that the prototype integrator, while not a compiler, still defines and translates a new language. It accepts collections of fragments of C code that must follow certain hard to formulate rules, which it then translates into C, but it does not accept C programs. Whether integration is performed by a compiler or by a simpler tool such as my prototype integrator, a new language is defined, and compilers can support better languages.

4.7 Barriers to Integration

There are situations in which integration is either not possible or not advantageous. Some are inherent to particular protocol functionalities, and some derive from the mapping of protocols onto system components such as address spaces. I have identified the following classes of barriers to integration:

Control Transfer If adjacent protocol layers are in different address spaces or different processors, transferring control between them is too slow for integration.

Message Reassembly Lost or out-of-order message fragments make it difficult for protocols above a reassembly layer to process isolated fragments.

Random Access Some protocols need random access to message data. They find the sequential access provided by a data stream too inefficient.

Retransmission In some cases a new physical copy of a message must be buffered, but whenever possible it is better to share a message buffer associated with a different protocol layer.

Runtime Protocol Path For some protocols, there is more than one possible next protocol, such as when demultiplexing.

There are strategies for minimizing barriers. The basic strategy is of course to avoid protocols (and protection domain boundaries) that are barriers. Granted that this is not always possible, barriers may be further minimized by a careful ordering of protocols in the protocol graph, and a careful mapping of protocols onto protection domains. The key observation is that barriers are only barriers if they occur between data manipulation protocols, since the performance advantage comes from integrating data manipulations, not arbitrary protocols. Protocols that do not manipulate data are only integrated when they are between data manipulation protocols. Hence if a barrier is located either above or below any data manipulations, then it does not prevent any integration. Also, if integration between two data manipulations has already been blocked by a barrier, then it does not matter if addition barriers are located between those two data manipulations; in effect, adjacent barriers coalesce into a single barrier.

4.7.1 Control Transfer Barriers

Serial and integrated implementations represent different tradeoffs between memory access and control transfer. Serial implementations minimize the number of control transfers—just one per message per layer—at the expense of the many memory accesses necessary to pass a large amount of data all at once. Integrated implementations attempt to minimize the number of memory accesses at the expense of frequent control transfers—in Morpheus, one for every data word (plus a few more) per layer. If control transfer is very slow, the optimal tradeoff is the purely serial implementation. Transferring control between protection domains (address spaces) is far slower than a function call, which is already much slower than the instruction sequencing we use to transfer control between integrated protocols when they pass words of data. Hence, a serial implementation is more efficient than integration across protection boundaries.

If part of a protocol stack is implemented on an outboard processor, integration between the host and outboard processor is similarly impractical, as is integrating a sending protocol stack on one host with a receiving protocol stack on another host. The problem is not strictly speaking control transfer, but rather synchronization; the synchronization overhead of passing data in small units is prohibitive.

Although it is inefficient to integrate protocols across such boundaries, it can be advantageous to integrate *boundary crossing mechanisms* with protocols on one side of the boundary. For example, the writing of a message into a buffer (or the reading of a message from a buffer) in a user address space can be integrated with protocols in the kernel address space. For another example, if a network device or outboard processor interface uses Programmed I/O, reading from or writing to those devices can be integrated with protocols (instead of using Programmed I/O to copy messages between devices and memory).

Control transfer barriers can be minimized by limiting the number of protection domains into which data manipulation protocols (and any system induced data manipulations such as copying across protection boundaries) are mapped. If all the data manipulations can be mapped into a single protection domain, then there is no control transfer barrier to prevent their integration.

Certain protocols may be pinned to particular protection domains due to their functionality or trust level. For example, the information necessary to perform a presentation formatting may be specific to an application, hence be pinned to the same user address space as the application. This could be accommodated by putting all the data manipulation protocols in that user address space. Any data manipulation protocols which would otherwise have been in the kernel address space, so as to be shared by the messages of applications in different user address spaces, would be migrated into the user address spaces. They would be either shared via shared libraries, or duplicated. Protocols below the first data manipulation protocol could remain in the kernel.

On the other hand, access to network devices must be restricted to the kernel to prevent unsafe access. Thus messages will still cross protection boundaries (at least logically), leaving the potential problem of boundary crossings themselves being data manipulations. For example, if a received message has to be transferred via DMA into memory for the kernel to determine the destination address space, then copying a message from the kernel address space into the destination address space would itself be a data manipulation.

This copy could not be integrated with the protocols in the user address space because it would require kernel privileges to access the message in kernel data space. Possible solutions include using copy-free techniques to logically move messages from one space to another, or using specialized network hardware to demultiplex messages directly to their destination address spaces [DAPP93]. If the data manipulation protocols must be distributed across kernel and user spaces, then copying messages across the user-kernel boundary can be integrated with kernel data manipulations at no additional cost.

4.7.2 Reassembly Barrier

From the perspective of higher layers, a message that is delivered to a reassembly protocol layer is not a message, but only a fragment of a message. It does not even contain the higher level headers unless the fragment is the first of a composite message. In general, protocols cannot process part of a message without having previously processed all the preceding parts of the message. Even if one could guarantee that fragments would always arrive in order and unduplicated, there would remain problems due to the possibly interleaved arrival of fragments of different messages. Higher level protocols would have to maintain state information for each composite message being processed, saving it between fragments and applying the corresponding state information to each fragment. It is not clear whether this can be accomplished while still supporting the efficient control transfers necessary for protocol integration. Gunningberg et al [GPSV91] report that when they attempted to integrate fragmentation/reassembly, their implementation schemes lost the layer abstraction. A general technique for integrating across reassembly layers independent of the particular protocols would be even more difficult to devise, if it is possible at all.

Fortunately, fragmentation/reassembly can be exiled to the bottom of the protocol graph, if it is needed at all. Gunningberg et al [GPSV91] believe that “for multimedia applications and gigabit networks we will see fragmentation only at the lowest layers or not at all,” and Clark and Tennenhouse’s Application Level Framing [CT90] also places fragmentation/reassembly at the bottom of the protocol graph. Application Level Framing proposes that a single notion of message framing—an Application Data Unit, essentially a unit of data that an application can process independently of other such units—be used throughout the protocol graph. Only at the bottom of the protocol graph are Application Data Units fragmented into, or reassembled from, the Transmission Data Units supported by the network.

4.7.3 Random Access Barriers

Some data manipulations involve essentially random access to message data. One example is the image transfer protocol described in [TP92]. This protocol's data manipulation reorders data to spread apart pixels that are adjacent in the original data, an image. The objective is to make it likely that if a pixel is lost, the adjacent pixels are not lost, and can therefore be used to compute an approximation to the value of the lost pixel. Consecutive pixels in the original image are far apart after the sending side data manipulation, and consecutive pixels in the incoming message are far apart in the resulting image after the receiving side data manipulation. Hence neither the sending nor receiving side of this protocol can efficiently manipulate data in a stream. (This particular example could however be implemented to accept a data structure message and output a stream, or vice versa.)

Another way of looking at random access barriers is that such protocols have a very large natural unit and cannot process anything smaller. The technique of accumulating input data in local state until a protocol's natural unit is available decreases in efficiency as the size of the natural unit increases. Clearly, for a natural unit on the order of the size of an entire message, a serial implementation would have better performance.

Protocols which utilize certain message editing operations may also be regarded as random access barriers. These operations are splitting a message into fragments, and clipping (discarding) part of a message from the tail end, as when stripping padding. A data structure implementation of messages can support these operations very efficiently without ever accessing the actual contents of messages [HMPT89, DAPP93], yet implementing such operations in terms of a data stream requires counting units of data to recognize the point in the message where the operation is to be applied.

4.7.4 Buffering for Retransmission

A protocol which buffers copies of messages for possible retransmission presents several problems. First consider retransmission. A retransmitted message can contain data, but originates, in effect, at a protocol layer, not an application. This is a problem because there could be data manipulations both above and below the retransmitting layer, but it is not possible to integrate the retransmission layer with preceding data manipulations since they should not process the retransmitted message. A Morpheus compiler could accommodate this by generating multiple integrated series of **sendThruput**s, one that starts above the retransmission layer to handle original transmissions, and one the starts just below the retransmission layer to handle retransmissions. The compiler would recognize the need to generate an extra integrated sub-series of protocols whenever it found a **sendThruput** in code other than the body of a **sendThruput**.

Buffering a copy of a message presents a more serious problem. Making a physical copy of a message, even if integrated, still entails writing the copy into memory. Also, unlike retransmission, which occurs only when a message is lost or late, buffering must take place for every message that is sent via a retransmission layer.

Buffering a physical copy of a message can often be avoided in serial implementations by retaining or sharing a preexisting copy of the message data structure. In the integrated case, one would likewise prefer to make a logical, or copy-on-write, copy whenever possible, i.e., whenever the buffering layer is situated relative to other layers such that the correct version of the message data was buffered for some other reason. These conditions hold except when the retransmission layer is included in an integrated series with read-write data manipulations both before and after it; otherwise, there is a message buffer (either the input or the output message) with the data in the correct form. These circumstances are determined at composition time, but that is too late since one needs to know at protocol design time because a single implementation of the retransmission protocol is not capable of supporting both logical and physical copying. Also, it is not clear how a retransmission protocol would obtain a reference to a message buffer associated with a layer that may precede or follow it by several layers. Solving these problems would likely complicate not only the compiler and the configuration software, but also the Morpheus source language.

Clearly retransmission layers complicate integration. If instead retransmission is treated as a barrier, it may be possible to simply avoid it. For example, it seems reasonable to locate any retransmission layers in the protocol graph so that they are above or below all data manipulations. A retransmission protocol located at the bottom of a protocol graph might be an appropriate optimization for a link known to have a high error rate. End-to-end retransmission at the application level might be appropriate because of the possibility of a protocol rejecting a message, and because alternatives to buffering and retransmission such as regenerating the data or just skipping the message may be appropriate depending on the application [SRC84].

4.7.5 Runtime Protocol Path Barriers

The protocol to which a given protocol will next pass a message might not be fixed at composition time. For example, demultiplexing determines at runtime which higher level protocol is to receive an incoming message. On the sending side, routing behaves similarly. Another example is message forwarding, since a message can either continue up the protocol graph if its destination is local, or be sent back down the protocol graph if it should be forwarded. Flow control in which a message can be blocked temporarily is another example, since a message may either continue through more layers or terminate, for the time being, at the flow control layer.

Determining a message's path through the protocol graph at runtime poses a problem for integration, at least for the implementation of integration thus far presented. This implementation combines all three stages of each protocol's **sendThruput** or **deliver-Thruput** in a single function. This implementation is "hard-wired" in the sense that, for a given operation and a given first protocol, the set of protocols that are integrated with it is fixed. Hence, this implementation can only integrate series of protocols which are known at compile time to be involved in processing any message received by the first in the series.

One response is to generate such integrated implementations for all the possible series of protocols that a message might traverse, and then select the correct one for a given message at runtime. This might use a mechanism such as the Packet Filter [MRA87] to determine the correct series of protocols. A packet filter is software can be programmed at runtime to interpret the combined protocol headers on an arriving message to determine the message's eventual destination. This approach could be used to anticipate the path that a particular message will follow through the protocol graph, and select the corresponding integrated groups of protocols. However, it does not address the generalized routing case or the flow control case. Furthermore, it adds redundant processing of message headers, and complicates protocols with the requirement to program the packet filter at runtime.

I propose a technique called *lazy messages* [2]. Essentially, it allows initial stages to be executed before binding a message to a particular integrated series of protocols. Each protocol's initial stage is implemented as a separate function. The initial stage must include any determination of the subsequent protocol. Each protocol's initial stage tags the message with the identity of that protocol, and attaches any information needed by the data manipulation or final stages of the same protocol, e.g. a checksum on an incoming message. When, in the serial execution of initial stages, a layer representing a barrier is reached, the protocol identifier tags on the message are used to select the function containing the corresponding integrated series of data manipulation and final stages. Such functions are constructed at compile time for each possible integrated series of protocols.

[2] The lazy message technique is based on a suggestion in [OP92].

These integrated series functions also include, for each layer, some initialization code which is responsible for initializing data manipulation filter variables. This may involve unpacking information attached to the message by the initial stage. This variable initialization code is executed in layer sequence before the combined data manipulation stage. Hence, in the lazy messages technique there are two "initial" stages, call them *initial-1* and *initial-2*. The bulk of the initial stage work is performed during the initial-1 stage, in particular the determination of the path through the protocol graph. The initial-2 stage is responsible for any initialization of data manipulation variables, such as the sum accumulated by a checksum.

The greater flexibility afforded by the lazy-message technique entails some costs. There is the increased complexity of separate initial-1 and initial-2 stages for the protocol programmer to deal with. Also, there is the extra space used by overlapping integrated series of protocols, since a function is constructed for each possible integrated series. However, the increased time to transfer control between stages of different layers and communicate information between stages of a given layer is not a significant cost, compared to the time to manipulate any significant amount of data.

Because of these costs, it might be better to simply use the hard-wired technique and attempt to minimize the instances of such barriers. The most common of these potential barriers is demultiplexing. Multiplexing at multiple levels in a protocol graph has other significant disadvantages unrelated to protocol integration, and therefore there should be only one multiplexing layer, at the bottom of the protocol graph [Ten89, Fel90]. This would have the side-effect of eliminating demultiplexing as a barrier. Similarly, any message forwarding could be performed at the bottom of the protocol graph, below any data manipulations.

4.8 Tradeoffs Between Performance and Abstraction

Morpheus's design for optimization makes tradeoffs of abstraction to gain performance. This tradeoff arises in two contexts: in protocol integration by itself, and in the existence of separate throughput-optimized and latency-optimized operations.

4.8.1 Trading Abstraction for Performance in Protocol Integration

The Morpheus technique for integrating protocols compromises clean abstraction to obtain low layering penalties; the optimization shows through at the source code level. Ideally, a compiler would automatically integrate operations expressed in the familiar serial style, but this seems to be considerably beyond current technology.

Morpheus supports integration in order to reduce the throughput penalty for layering. Integration in Morpheus could be regarded as simply improving performance, but I choose to view it as reducing a performance penalty for protocol modularity; since a non-modular, or monolithic, implementation of a protocol suite is in theory free to integrate protocols without concern for module boundaries, any inability to integrate in a layered implementation may be regarded as a penalty for layering. A reduced penalty for layering makes it practical to decompose network software into building blocks protocols, and also supports a high level of abstraction by permitting decomposition to the granularity required by the shapes constraint. Thus protocol integration in Morpheus supports protocol development.

Unfortunately, Morpheus integration also compromises abstraction, thereby undermining support for protocol development to some extent. Without integration, the programmer would implement a **sendThruput** or a **deliverThruput** as a single intuitive function; but with integration, there are distinct operations for control and data messages, and the **sendThruput** and **deliverThruput** operations are programmed as sets of functions with relatively unintuitive purposes and interrelationships.

This design choice in Morpheus—to favor performance at the expense of clean abstraction—is based on the principle that performance is the overriding concern for network software designers. If there is a significant performance cost for some other benefit, then they will sacrifice that benefit in favor of performance.

4.8.2 Separate Latency-Optimized and Throughput-Optimized Operations

Morpheus imposes on the programmer the burden of coding separate latency-optimized and throughput-optimized operations. This represents another tradeoff of clean abstraction in favor of performance. There are a number of possible alternatives to this design.

The first alternative to consider is that a single operation might be optimized for both latency and throughput— that all the optimizations could be combined in one **send** and one **deliver**. Suppose an operation supported the protocol integration (throughput) optimization. The latency optimizations of generating utility operations inline and eliminating header bounds checking can also be applied (and are applied to both sets of operations in the current design). However, the remaining latency operations are not appropriate. Dedicated message registers and short-circuit return apply to function calls between layers (since they are designed to support runtime configuration), but protocol integration combines operations from adjacent protocols into a single function. They could however be applied at protocol boundaries where lazy messages are used to determine the sequence of protocols. The remaining latency optimization, procedure cloning, could be applied to this operation, but space costs argue against it. Both procedure cloning and protocol integration multiply space costs by duplicating code. Supporting them in separate operations results in space usage equal to the sum of their individual space usages, while providing the appropriate optimization to each message; while combining them in the same operation results in space usage which is the product of their individual usages, even though each message only benefits from one or the other optimization.

One might ask whether a throughput-optimized operation also has good latency despite the limited applicability of the latency optimizations, especially since protocol integration combines operations from adjacent layers into a single function. Unfortunately, for some protocols latency would be hurt by the task-level ordering discipline, e.g. deferring external behavior to the final stage, or by the requirement to structure data manipulations as word filters. Latency would be even more harmed anywhere lazy messages are used due to traversing layers twice (once for the initial stage, and once for the remainder), marshalling and demarshalling any information that must be attached to the message to communicate it from the initial stage to the subsequent stages (e.g. the checksum from a received message), and using protocol tags on messages to look up the function corresponding to the remaining stages of the correct protocols.

If a single operation cannot be effectively optimized for both latency and throughput, the next question to ask is whether separate operations might be generated from the same source code. Unfortunately, that shared source code would need an awkward syntax like that of the current throughput-optimized operations to support protocol integration. Furthermore, the control message path could suffer some increased latency due to the requirement to structure the source code to support word filters and the task-level ordering discipline.

A number of reasonable compromises to simplify programming are possible:

- Integration-oriented source code could be used to generate both operations by default, with the programmer having the option of providing separate source code for the latency-optimized operation if desired.

- The programmer could opt to provide source code only for the latency-optimized operation, making that layer into a barrier to integration.

- Both operations could optionally share a single source code level expression of a data manipulation as a word filter.

- Where the data manipulation applied to the higher level headers is the same as that applied to the application data, both manipulations could optionally be generated from a single source code word filter.

- The programmer could be relieved of having to code any data manipulation for latency-optimized operations by applying the latency-optimized operations only to messages which contain no application data. This would have the performance disadvantage of optimizing messages with small amounts of data for throughput when they should be optimized for latency.

CHAPTER 5

CONCLUSIONS

My thesis is that the combination of two novel strategies, constraining protocol specifications and using a special-purpose language, provides powerful program development support for network software. In support of this thesis, I have shown that these strategies enable Morpheus to achieve three well-established principles of software development: abstraction, modularity, and software reuse. A key intermediate step is Morpheus's optimizations that reduce the performance penalty for protocol layering.

This chapter summarizes the specification-level constraints that have been proposed as a result of this work, summarizes the contributions of this research, and discusses future work.

5.1 Summary of Constraints

Morpheus imposes the following constraints on protocol specifications:

- Message headers and application data must each use an integral number of words, and header fields must be word-aligned relative to the start of their headers. This ensures that all header fields can be word-aligned in memory for efficient access (Chapter 2), and that application data can be manipulated using word filters (Chapter 4). Headers and application data can be padded to satisfy this constraint.

- Protocols must not support runtime options (Chapter 2). Eliminating runtime options increases reusability of protocol implementations since otherwise protocols could be composed only if each protocol that invoked an option were paired with a protocol that implemented the option. Different options should instead be implemented as distinct protocols. Where the correct option would depend on runtime information, a router can be used to route messages through the appropriate protocol.

- Protocols must not provide functionality corresponding to more than one shape (Chapter 2). This allows Morpheus to automatically provide the aspects of a protocol's implementation that are determined by a protocol's declared shape. Protocol functionality that spans more than one shape can be implemented as the composition of multiple protocols.

- Multiplexing must use two multiplexing keys, and their types must be Morpheus's standard multiplexing key type (Chapter 2). This allows Morpheus to automatically provide the implementation of multiplexing and demultiplexing, and simplifies configuring multiplexors in a protocol suite. To behave like conventional multiplexing based a single key, a multiplexor can be configured so that the local and remote multiplexing keys are identical.

- Protocols must be able to manipulate higher level headers and application data separately, and network drivers (or software between the Morpheus protocol subsystem and the network drivers) must support the segregated message abstraction (Chapter 4). This allows efficient protocol integration by providing protocols with a common definition of the data they are to manipulate together. This constraint rules out only those protocols that apply a single manipulation to both higher level headers and application data, and define that manipulation in terms of a unit that could span the boundary between the two.

5.2 Contributions

The main contributions of this research are the protocol abstractions and the optimization techniques.

Morpheus's protocol abstractions provide a high level of abstraction that supports protocol development by providing a seamless model for thinking about protocols and relieving the programmer of making and expressing low-level design decisions that are unnecessary and may introduce errors or poor performance. Protocol shapes provide a prescription for decomposing network software while raising the level of abstraction. The abstractions also present a standardized protocol interface that is well-suited to decomposition, thereby supporting protocol development through the building-blocks approach.

Morpheus's procedure cloning technique uses knowledge of a restricted domain to extend a powerful compile-time interprocedural optimization to a runtime situation, while simultaneously avoiding the overhead of interprocedural analysis. Morpheus's dedicated message registers and short-circuit return techniques support runtime configuration of protocol suites by improving performance in contexts where compile-time techniques cannot be applied.

Morpheus contributes a protocol integration technique that is very general and preserves modularity. The same basic technique can be used to integrate not only reusable building-blocks protocols, but also more conventional protocols. Even when protocol implementations are not expected to be reused, the modularity supported by this technique is a considerable advantage. Even if implementors choose to dispense with modularity, word filters, segregated messages, and the task-level ordering discipline provide a very general technique for implementing integrated protocols. And even if implementors intend to customize an implementation in order to obtain the greatest possible performance, this technique can serve as the basis for the customized implementation.

Identifying the four protocol integration problems is a contribution. Researchers in search of better general solutions can use them as a starting point. Implementors of custom integrated protocol suites can use them to better understand the specific problems they encounter in the context of particular suites.

Finally, the experiments and analysis of Morpheus's integration technique quantify ILP behavior and potential, and provide a lower bound on the performance benefit that can be obtained. If improvements in processor performance continue to outpace improvement in memory performance as expected, the performance benefit of integration will only increase relative to measurements on current machines.

5.3 Future Work

The research presented in this dissertation is highly exploratory. A great deal of work could be invested simply to complete the realization of Morpheus: a complete language design, with fully-specified syntax and at least a fairly rigorous semantics, if not a formal semantics; working compilers for one or more target machines; and thorough performance measurements based on compiler-generated code. Other language environment work would include support for protocol-oriented debugging.

The Morpheus design presented here supports only the asynchronous, peer-to-peer (unicast) protocols. It should be extended to such other forms of communication as Remote Procedure Call and multicast communication.

There is the potential to improve throughput significantly by integrating data manipulations performed by applications with those performed by protocols. It is not clear how to achieve this in Morpheus since Morpheus can only express protocols.

A protocol implementation language would seem to be well-suited to support multiprocessing implementation of protocols. It may be possible for a compiler for such a specialized language to transparently generate the appropriate locking, so that protocol source code is independent of the degree and style of multiprocessing. A compiler would know the target multiprocessing system, have access to information about which data is shared, and have the ability to interject locking code at arbitrary points. If this is not possible—if it were to turn out that the programmer must be aware of multiprocessing—

there remains the potential to provide protocol-oriented multiprocessing abstractions of a higher level than the basic lock.

5.4 Concluding Remarks

The underlying theme of this research is that the building-blocks approach to building network software works. Decomposing network software into small, reusable protocols makes it easier to understand and develop, especially when combined with the strategies employed by Morpheus. Protocol-oriented optimizations give building-block protocols a potential performance nearly that of more monolithic implementations, and the ease of development allows programmers to come closer to achieving this potential performance than their counterparts who build monolithic implementations.

The main impediment to the building-blocks approach is the continued dominance of such current network architectures as TCP/IP. However, factors discussed in Chapter 1 are leading to the obsolescence of these architectures. My hope is that the work presented in this dissertation will influence the design and implementation of the network software that supersedes them.

APPENDIX A
C VERSION OF SEQUENCER PROTOCOL

This is a C implementation of the protocol SEQUENCER. It illustrates the level of detail that is necessary for a complete protocol implementation in a general purpose language.

```
#include "bunchOfStuff.h"

/* Header */
typedef struct{
        int    seqNum;
}*Hdr;
#define NET_BYTE_ORDER    LITTLE_ENDIAN

/* Protocol */
typedef struct{
        String      name;
        ProtlOps    ops;
        Sap         undrSap;
        Sap         overSap;
        int         sendSeqNum;
}*Protl;

/* OverSessn */
typedef struct{
        GenericUndrSessn  otherSide;
        Sap               sap;
        Protl             protl;
        Pfv               send;
        Pfv               grantDelivs;
        struct undrSessnStruct*   undrSessn;
}*OverSessn;

/* UndrSessn */
typedef struct undrSessnStruct{
        GenericOverSessn  otherSide;
        Sap               sap;
        Protl             protl;
        Pfv               deliv;
        Pfv               grantSends;
        OverSessn         overSessn;
        int               rcvSeqNum;
```

```
}*UndrSessn;

static void send( overSessn, msg )
OverSessn     overSessn;
Msg*          msg;
{
Hdr           hdr;
int           seqNumInNetByteOrder;

        hdr = (Hdr) msgPush( msg, sizeof( Hdr* ) );
        seqNumInNetByteOrder =
              hostToNetInt( NET_BYTE_ORDER, overSessn->protl->sendSeqNum++ );
        bcopy( &seqNumInNetByteOrder, &header->seqNum, sizeof(int) );
        overSessn->undrSessn->otherSide->send(
              overSessn->undrSessn->otherSide, msg ) );
}

static void deliv( undrSessn, msg )
UndrSessn     undrSessn;
Msg*          msg;
{
Hdr           hdr;
int           seqNumInNetByteOrder;
int           seqNumInHostByteOrder;

        hdr = (Hdr) msgTop( msg, sizeof( Hdr* );
        bcopy( &header->seqNum, &seqNumInNetByteOrder, sizeof(int) );
        seqNumInHostByteOrder =
              netToHostInt( NET_BYTE_ORDER, seqNumInNetByteOrder );
        if( seqNumInHostByteOrder > undrSessn->rcvSeqNum ){
              undrSessn->rcvSeqNum = seqNumInHostByteOrder;
              msgPop( msg, sizeof( Hdr* ) );
              undrSessn->overSessn->otherSide->deliv(
              undrSessn->overSessn->otherSide, msg ) );
        }else{
              undrSessn->otherSide->grantDelivs( undrSessn->otherSide, 1 );
        }
}

static void grantSends( overSessn, numCredits )
OverSessn     overSessn;
int           numCredits;
{
        overSessn->undrSessn->otherSide->grantSends(
              overSessn->undrSessn->otherSide, numCredits );
}

static void grantDelivs( undrSessn, numCredits )
UndrSessn     undrSessn;
int           numCredits;
```

```
{
	undrSessn->overSessn->otherSide->grantDelivs(
		undrSessn->overSessn->otherSide, numCredits );
}



static void createOverSessnDown( overSap, higherUndrSessn, addr )
Sap		overSap;
GenericUndrSessn	higherUndrSessn;
GenericAddr	addr;
{
OverSessn	overSessn;
UndrSessn	undrSessn;

	/* create a sequencer overSessn and glue to the higher level undrSessn */
	overSessn = (OverSessn) malloc( sizeof(OverSessn*) );
	overSessn->sap = overSap;
	overSessn->send = send;
	overSessn->grantDelivs = grantDelivs;
	overSessn->otherSide = higherUndrSessn;
	higherUndrSessn->otherSide = overSessn;

	/* set up the corresponding sequencer underSessn */
	undrSessn = (UndrSessn) malloc( sizeof(UndrSessn*) );
	((Protl)overSap->lowerProtl)->undrSap->createOverSessnDown(
		((Protl)overSap->lowerProtl)->undrSap, undrSessn, addr );
	undrSessn->rcvSeqNum = 0;

	/* link corresponding sequencer overSessn and undrSessn */
	undrSessn->overSessn = overSessn;
	overSessn->undrSessn = undrSessn;
}

static void enableUpwardSessnCreate( overSap )
Sap		overSap;
{
	((Protl)overSap->lowerProtl)->undrSap->enableUpwardSessnCreate(
		((Protl)overSap->lowerProtl)->undrSap );
}

static void createUndrSessnUp( undrSap, lowerOverSessn, addr )
Sap		undrSap;
GenericOverSessn	lowerOverSessn;
GenericAddr	addr;
{
UndrSessn	undrSessn;
OverSessn	overSessn;

	/* create a sequencer undrSessn and glue to the lower level overSessn */
```

```
	undrSessn = (UndrSessn) malloc( sizeof(UndrSessn*) );
	undrSessn->sap = undrSap;
	undrSessn->deliv = deliv;
	undrSessn->grantSends = grantSends;
	undrSessn->otherSide = lowerOverSessn;
	lowerOverSessn->otherSide = undrSessn;
	undrSessn->rcvSeqNum = 0;

	/* set up the corresponding sequencer overSessn */
	overSessn = (OverSessn) malloc( sizeof(OverSessn*) );
	((Protl)undrSap->higherProtl)->overSap->createUndrSessnUp(
		((Protl)undrSap->higherProtl)->overSap, overSessn, addr );

	/* link corresponding sequencer overSessn and undrSessn */
	overSessn->undrSessn = undrSessn;
	undrSessn->overSessn = overSessn;
}

static GenericAddr getLocalAd( overSap )
Sap	overSap;
{
	return( ((Protl)overSap->lowerProtl)->undrSap->getLocalAd(
		((Protl)overSap->lowerProtl)->undrSap ) );
}

static void addOverSap( protl, overSap )
Protl	protl;
Sap	overSap;
{
	protl->overSap = overSap;
	overSap->createOverSessnDown = createOverSessnDown;
	overSap->enableUpwardSessnCreate = enableUpwardSessnCreate;
	overSap->getLocalAd = getLocalAd;
}

GenericProtl sequencerInitProtl( undrSaps )
Saps	undrSaps;
{
Protl	protl;

	protl = (Protl) malloc( sizeof( *Protl ) );
	protl->undrSap = undrSaps[0];
	protl->undrSap->higherProtl = (GenericProtl) protl;
	protl->undrSap->createUndrSessnUp = createUndrSessnUp;
	protl->sendSeqNum = 1;
	return( (GenericProtl) protl );
}
```

REFERENCES

[And85] David Anderson. *A Grammar-Based Methodolody for Protocol Specification and Implementation*. PhD thesis, University of Wisconsin - Madison, August 1985.

[Cho85] T. Y. Choi. Formal techniques for the specification, verification and construction of communication protocols. *IEEE Communications*, October 1985.

[CHT91] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 91.

[CJRS89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[Cla82] David D. Clark. Modularity and efficiency in protocol implementation. Request for Comments 817, MIT Laboratory for Computer Science, Computer Systems and Communications Group, July 1982.

[Cla85] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 171–180, December 1985.

[Com88] Douglas Comer. *Internetworking with TCP/IP*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Coo83] Keith D. Cooper. *Interprocedural Data Flow Analysis in a Programming Environment*. PhD thesis, Rice University, April 1983.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, September 1990.

[DAPP93] Peter Druschel, Mark B. Abbott, Michael A. Pagels, and Larry L. Peterson. Network subsystem design: A case for an integrated data path. *IEEE Network Magazine*, July 1993.

[Fel90] David C. Feldmeier. Multiplexing issues in communication system design. In *Proceedings of the SIGCOMM '90 Symposium*, 1990.

[GNI92] Murray W. Goldberg, Gerald W. Neufeld, and Mabo R. Ito. The parallel protocol framework. Technical Report 92-16, Department of Computer Science Department, University of British Columbia, August 1992.

[GPSV91] Per Gunningberg, Craig Partridge, Teet Sirotkin, and Bjorn Victor. Delayed evaluation of gigabit protocols. In *Proceedings of the 2nd MultiG Workshop*, 1991.

[Hal91] Mary W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, April 1991.

[HMPT89] Norman C. Hutchinson, Shivakant Mishra, Larry L. Peterson, and Vicraj T. Thomas. Tools for implementing network protocols. *Software—Practice and Experience*, 19(9):895–916, September 1989.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[HPAO89] Norman C. Hutchinson, Larry L. Peterson, Mark Abbott, and Sean O'Malley. RPC in the *x*-Kernel: Evaluating new design techniques. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 91–101, December 1989.

[KR86] Brian W. Kernighan and Dennis M. Ritchie. The m4 macro processor. In *Unix Programmer's Supplementary Documents Volume 1*. University of California at Berkeley, April 1986.

[MRA87] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 39–51, November 1987.

[OMa90] Sean W. OMalley. *Avoca: An Environment for Programming with Protocols*. PhD thesis, University of Arizona, August 1990.

[OP92] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Pia83] T. F. Piatkowski. Protocol engineering. In *Proceedings of 1983 International Communications Conference*, June 1983.

[PMI88] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, winter 1988.

[Rud85] H. Rudin. An informal overview of formal protocol specification. *IEEE Communications*, March 1985.

[Saj85] M. Sajkowski. Protocol verification techniques: Status quo and perspectives. In *Protocol Specification, Testing, and Verification, IV*, 1985.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, MA, 1986.

[Tan88] Andrew S. Tannenbaum. *Computer Networks*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Ten89] David L. Tennenhouse. Layered multiplexing considered harmful. In *Proceedings of the 1st International Workshop on High-Speed Networks*, November 1989.

[TP92] Charles J. Turner and Larry L. Peterson. Image transfer: An end-to-end design. In *Proceedings of the SIGCOMM '92 Symposium*, Baltimore, Maryland, August 1992.

[TT87] American Telephone and Inc. Telegraph. *Unix System V Streams Programmer's Guide*. Prentice Hall, Englewood Cliffs, NJ, 1987.

[vB87] Gregor v. Bochmann. Usage of protocol development tools: The results of a survey. In *Protocol Specification, Testing, and Verification, VII*, 1987.