A Language-Based Approach to Protocol Implementation

Mark B. Abbott and Larry L. Peterson[1]

TR

Abstract

Morpheus is a special-purpose programming language that facilitates the efficient implementation of communication protocols. Protocols are divided into three categories, called *shapes*, so that they can inherit code and data structures based on their category; the programmer implements a particular protocol by refining the inherited structure. Morpheus optimization techniques reduce per-layer overhead on time-critical operations to a few assembler instructions even though the protocol stack is not determined until runtime. This supports divide-and-conquer simplification of the programming task by minimizing the penalty for decomposing complex protocols into combinations of simpler protocols.

September 23, 1992

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

[1]This research supported in part by DARPA Contract DABT63-91-C-0030.

1 Introduction

Network software is difficult to design and implement. As with any distributed concurrent program with complex functionality, correctness is difficult to achieve. This situation is exacerbated by the additional requirement of high performance. This paper introduces a new approach to the network software problem—using a programming language designed specifically for high performance protocol implementations. We have designed such a language, called Morpheus.

The foremost advantage of a programming language is that it is an ideal vehicle for both imposing and benefiting from constraints. In other words, a language provides both the means to restrict the design choices available to the programmer, and the medium in which to realize the advantages due to the narrower design domain. The constraints imposed by Morpheus take the form of strategies and techniques carefully selected from among those exposed by collective experience with networks [2, 11, 18], plus a novel constraint (*shape*) introduced in this paper.

Stated another way, there are two motivations for the constraints imposed by Morpheus. First, they enforce a good design discipline. It has been argued that the development of a new engineering discipline often happens in two phases [9]. 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. Morpheus is a tool of the second phase: it is a special purpose programming language that provides an explicit model for thinking about and concisely expressing protocols in accordance with a design discipline.

The second motivation for Morpheus constraints is that it makes possible a more powerful tool. In effect, the more the user is constrained, the more the tool knows about what the user wants to do. For example, protocols written in Morpheus can be compiled into more efficient object code than those written in general purpose languages because a Morpheus compiler has more domain knowledge available to apply to low level optimization.

The constraints imposed by Morpheus are not confined to the internal implementation details of a protocol, but also extend to externally visible factors such as message header format. In other words, Morpheus preempts design choices not only from the final implementors of protocols, but also from designers and standards committees. Consequently, *an existing protocol specification may not be implementable in Morpheus.* This result is to be expected of an approach based on constraining the design space. If Morpheus were restricted to those constraints which are satisfied by all existing protocols, it would lose much of its power. Hence, by its very nature, our approach cannot concern itself with existing protocols.

Note however that the inability to satisfy a particular specification doesn't imply the inability to implement the corresponding functionality. On the contrary, *any communication semantics can be implemented in Morpheus*. The functionality of a given existing protocol is likely to be realized by a composition of several Morpheus protocols, or at least involve using a different message syntax.

Morpheus adopts an object-oriented programming style. In Morpheus, the fundamental protocol abstractions are represented as objects. The Morpheus programmer implements a particular protocol by refining the code and data structures inherited from pre-defined base classes. The shape constraint serves to maximize the amount of code and data structures that can be inherited.

Morpheus optimizations reduce per-layer overhead on time-critical operations to a few assembler instructions, even though the protocol graph is not determined until runtime. In addition to the immediate performance benefit, this supports divide-and-conquer simplification of the programming task by minimizing the penalty for decomposing complex protocols into combinations of simpler protocols. Simpler protocols also have greater potential for reuse.

Morpheus is more than a protocol model added to an existing language. Morpheus performs optimizations that existing languages cannot because they lack Morpheus' built-in knowledge of the common patterns of use of the elements of the model. It also manages the machine-dependent aspects—alignment and byte order—of message header manipulation. Although we have not yet exploited it, Morpheus might also be used to hide the granularity of concurrency on a multiprocessor.

This paper makes two important contributions. First, it describes the Morpheus protocol architecture and programming model, most notably the shape constraint. Second, it discusses the optimization techniques available to the Morpheus compiler. While no Morpheus compiler has as yet been constructed, we document the effectiveness of these techniques with hand-coded optimizations in assembler language for the MIPS R3000 architecture [12].

2 Background

Morpheus was not designed in a vacuum. It reflects our understanding of network software based on experience building tools to support the rapid implementation of efficient protocols. This section identifies related work that has influenced Morpheus, discusses the biases in its design, and identifies certain limitations.

2.1 Related Work

Generally speaking, there are two common approaches to implementing network protocols. At one extreme, the protocol is written in a general purpose language, subject only to the constraints imposed by the operating system and architectural environment in which the protocol will be used. Although protocols implemented in this way are generally efficient, the programming task can be exceptionally difficult, depending on the extent to which the host operating system is designed to accommodate network protocols. In the best case, the operating system provides explicit support for implementing protocols. For example, the *x*-kernel provides a uniform protocol interface and a protocol support library [11].

At the other extreme, network software is automatically derived from a protocol specification expressed using a Formal Description Technique (FDT) such as Estelle, LOTOS, or SDL [17]. In its current state, however, this technology has not lived up to its promise. Instead of expressing a protocol in purely abstract terms, protocols are specified in relatively "implementation-oriented" FDTs, and the software generated is generally in the form of a skeleton which must be completed with programmer code.

This paper introduces a language-based approach to protocol implementation that lies between these two extremes. Our language-based approach has several advantages. First, it affords the opportunity to perform low level optimizations specific to protocol implementations. Second, code and data structures that are routine parts of protocols can be automatically provided, just as procedure prologue and epilogue code is automatically generated by general purpose languages. Third, a language can present a seamless model of protocols—no underpinnings are visible below the single level of abstraction.

Our approach is not independent of the two alternatives; it is related to the OS-based and the FDT-based approaches as follows.

We were led to this approach by our experience with the *x*-kernel, which takes the OS-based approach. Specifically, we realized there were several ways in which we wanted to extend the *x*-kernel that could best be accomplished by embedding *x*-kernel-style protocol abstractions in a language. Morpheus is the result: it offers a more convenient way to express protocols by replacing the "boilerplate" found in *x*-kernel protocols with automatically generated code, it implements the operating system aspects of the *x*-kernel in the language's runtime system, and it takes advantage of compile-time optimizations not available in a

general purpose programming language. A considerable additional advantage which we have not pursued is the potential for a Morpheus debugger that would share Morpheus' knowledge of the structure and behavior of protocols.

Morpheus is like FDTs in that it strives to present a higher level of abstraction to the programmer/specifier. However, Morpheus' level of abstraction is not as high as that of FDTs; programmers still write in a familiar, imperative programming style. Furthermore, Morpheus is far less general, imposing constraints that eliminate design choices. But it is just this combination that makes it possible to present the programmer with the right design options and a relatively high level of abstraction from which complete, high-performance implementations can still be generated with current software technology.

Morpheus represents a middle ground between these two approaches to protocol design. It retains the performance orientation of its roots in operating system support, but strives for a level of abstraction closer to that of formal specifications. Hence, Morpheus might represent a good target language for automatic synthesis of protocol implementations from formal specifications. Assuming that a given protocol specification were consistent with Morpheus constraints, synthesizing an implementation would only require translating the specification to the level of Morpheus abstractions, whereupon the Morpheus compiler would provide an efficient implementation of those abstractions.

2.2 Architectural Biases

One of the liberties we have taken with this research is to imagine what network software could look like if not constrained by today's standardized protocols. Current protocols often include artifacts that are not fundamental to network communication and that interfere with innovations. While we are interested in providing the same communication services (semantics) as are available in today's networks, we do so without regard for the exact form (syntax) of today's protocols.

In particular, Morpheus supports a dynamic network architecture, such as the one described in [13]. This architecture has two key characteristics:

- There are many, very simple protocols.
- Protocols are selected and combined at runtime.

These characteristics have ramifications that are apparent in the design of Morpheus. Consider the following two biases.

First, Morpheus has a bias towards composing network software from the simplest possible protocols, going so far as to *require* a certain degree of simplicity. This has decisive advantages when compared to using shorter stacks of more complex protocols. The primary advantage is that of divide-and-conquer: a complex protocol is harder to develop, verify, implement, and maintain than an equivalent collection of simple protocols. A second advantage is that of reuse. Reuse of protocol implementations requires not only that interfaces are syntactically compatible, but also that the composition of semantics is useful. Large complex protocols are unlikely to implement the precise combination of functions that is appropriate in another context. A third advantage of simpler protocols is the increased potential (exploited by Morpheus) for automatically generating parts of an implementation.

Additional advantages of simple protocols emerge when they can be selected and combined into a protocol stack, or more generally, a *protocol graph*. For one thing, a communication service user can configure exactly the communication service (i.e. protocol graph) needed, instead of sharing a statically determined graph. Also, the development of new protocols is easier because binding decisions are delayed.

Finally, it is easier to adapt to changes in the underlying technology; i.e., substitute new protocols that are better suited for the new technology.

Morpheus' second bias is against logical multiplexing—the combining by a protocol of multiple higher-level data streams into a single lower-level data stream. This reflects the growing recognition of the penalties for logical multiplexing [7, 16]:

- When streams are merged, they can't be distinguished for purposes of quality of service.
- Multiplexing and demultiplexing at multiple layers hurts performance by duplicating effort.
- Multiplexing is a barrier to the propagation of flow and congestion control information between protocol layers.

Morpheus does not assume that every protocol multiplexes; in fact, Morpheus relegates the multiplexing protocols to a special category of protocol. Furthermore, Morpheus' inclusion of an interface for propagating flow control information between layers is predicated on the assumption that relatively few layers multiplex.

2.3 Limitations

Given its novel approach to a problem area of enormous breadth, the work presented in this paper must be viewed as exploratory. In particular, we identify the following limitations of our work.

First, we have restricted the problem domain. Morpheus supports only asynchronous, peer-to-peer (unicast) protocols. While this is the most fundamental class of protocols, multicast or group communication, and synchronous communication including Remote Procedure Call (RPC) are clearly very important. Although not discussed in this paper, we have considered how to support such different classes of protocols in a single framework, and have worked out in some detail the extensions required for RPC.

Second, we have not as yet implemented a Morpheus compiler. Although we are able to report some preliminary performance measurements based on hand optimizations of an implementation of the Morpheus protocol architecture in C, the lack of a compiler is a considerable handicap. One consequence is that we are unable to make definitive statements about how the language should be implemented, and in particular, which aspects of the language would be best implemented via code generation, versus inheritance, versus shared infrastructure. Thus, any discussion of the implementation presented in the paper should be viewed as describing one possible implementation strategy rather than the best or only strategy. Another consequence of not having implemented a compiler is that we cannot be sure that the optimizations proposed in Section 5 will not suffer from phase conflicts with conventional optimizations.

A third limitation of this work is that we do not have a wealth of experience implementing protocols in Morpheus. This is primarily due to the lack of a compiler. This is not to say that we have no experience base upon which to claim that Morpheus' design is a reasonable one—-Morpheus' design follows directly from our experience implementing protocols in the *x*-kernel. One could view this work as a branch in the evolution of the *x*-kernel, in which each mutation was considered in the context of (and often directly motivated by) *x*-kernel experience.

Finally, while our underlying goal with Morpheus is to influence the design and implementation of protocols in the future, in the near term Morpheus appears to have limited applicability since it is generally incompatible with the protocols that are entrenched in the current network world. However, one feasible near-term use of Morpheus is implementing application-specific protocols. That is, one could depend on standard network and data link protocols to deliver packets to the end host, quickly demultiplex them to the

destination address space, and then use Morpheus to implement the transport and higher-level protocols on an application by application basis.

3 Language Abstractions

There are two main design goals behind Morpheus' language abstractions. First, the fundamental network abstractions such as messages and connections should be an integral part of the language. Second, Morpheus should automatically supply the predictable code and data structures appropriate for a given protocol. This second goal grew out of our experiences with the *x*-kernel. As is often the case in writing software, one programs an *x*-kernel protocol by first copying the code from a similar protocol to use as a template, and then editing that code to obtain the desired protocol. This approach derives its benefit from the fact that there are routine tasks, such as manipulating headers and demultiplexing, that are common to many protocols. Morpheus has the goal of performing the equivalent task automatically.

Note that automatically supplying code constrains the programmer in that it preempts design choices, reducing certain design options to a single "option" for which Morpheus can supply the code. The obvious benefit is that the programmer doesn't have to write that code. The less obvious benefit is that in these cases, *the programmer can't make a bad design choice*.

3.1 Object-Based Design

Morpheus represents the fundamental protocol abstractions as objects. Morpheus pre-defines a collection of base classes, and the programmer implements a protocol by refining these base classes to produce subclasses that are appropriate to the specific protocol, as illustrated in Figure 1. An instance of a protocol is made up of objects which are instances of the subclasses specific to that protocol. Representing protocol abstractions this way not only achieves our goal of making the fundamental abstractions explicit in the language, but it also supports our second goal by providing a language-level mechanism—*inheritance*—for supplying pre-defined code and data structures.

Morpheus' model of protocols divides them into categories, called *shapes*, based on their functionality. Shapes are a particularly novel aspect of Morpheus. By constraining protocols to conform to these shapes, Morpheus gains additional information about each protocol which it uses to supply more code than would be possible for arbitrary protocols.

In order to supply shape-based code via the mechanism of inheritance, the shape dimension must be integrated with the class hierarchy. Figure 2 schematically depicts the relationship between classes and shapes. The abstract classes at the top of the hierarchy specify interfaces used by all protocols. Each base class at the middle level of the hierarchy adds code and data structures appropriate for all protocols of the corresponding shape. Each subclass at the bottom of the hierarchy adds the remaining code and data structures to complete the implementation of a specific protocol.

Object-oriented programming is well suited for representing our protocol abstractions. One characteristic of our model of protocols is that it 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.

Another benefit is notational economy. Procedures implementing operations on an object (the *self* object) can refer to the state variables of that object directly, without explicit reference to the object. This benefit is multiplied in Morpheus because the ability to directly refer to state variables is extended to those objects of

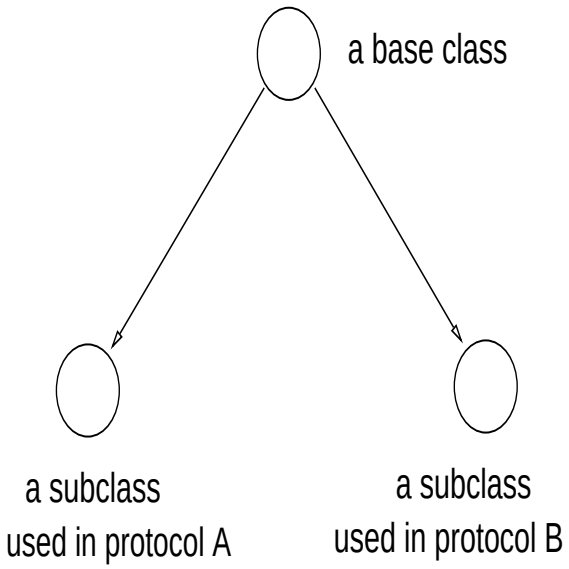

Figure 1: Protocols as Refinements

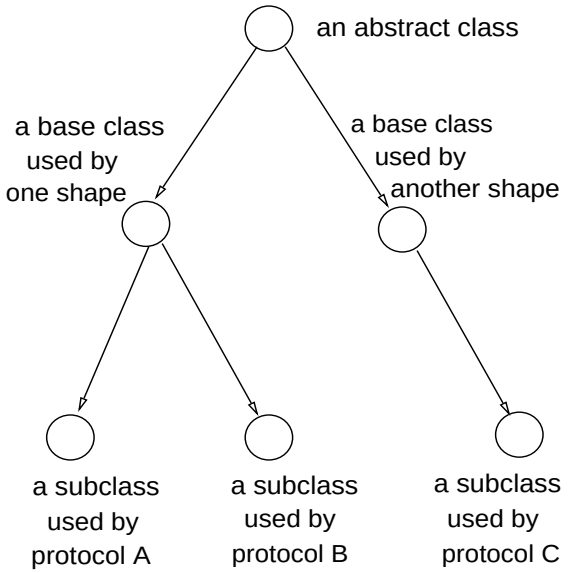

Figure 2: Classes and Shapes

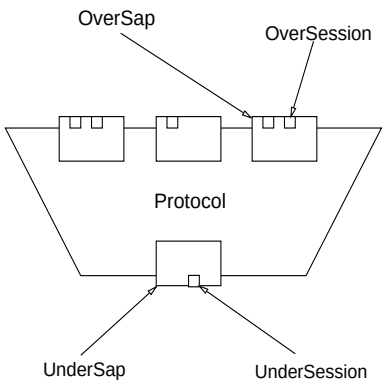

Figure 3: Base Classes

which the self object is a component. In other words, some Morpheus objects are contained in other objects, in which case they can refer to the state variables of those containing objects.

A final benefit is that object-oriented programming provides inheritance as a language-level discipline for automatically supplying behavior and data structures. Alternative techniques that are outside the language—such as macros or library routines—might be workable, but their benefit would be offset by the burden of working with mechanisms outside the language.

It should be made clear that Morpheus is object-oriented only with respect to the built-in protocol abstractions; the Morpheus programmer cannot define completely new classes. Also note that Morpheus' benefits could not be duplicated by adding pre-defined classes to a general object-oriented language such as C++ [15] since it would lack the knowledge of common patterns of protocol operation invocation that Morpheus exploits to optimize. Also, a general purpose language could not manage alignment and byte order considerations for message headers. Finally, certain syntactic niceties would have to be sacrificed.

3.2 Base Classes

Morpheus defines five programmer-refined base classes, corresponding to the fundamental elements of Morpheus' model of protocols. The Morpheus programmer implements a protocol by refining the base classes, thereby deriving subclasses that are specific to the protocol. A subclass is derived from a base class by adding new state information (declaring additional instance variables) and by modifying and extending the base class behavior (defining additional procedure code that augments or overrides the base class procedures).

The five base classes—*Protocol*, *OverSap*, *UnderSap*, *OverSession*, and *UnderSession*—are schematically depicted in Figure 3. Note that some objects are nested in others; a Protocol object includes as components some OverSap and UnderSap objects, OverSaps in turn have OverSessions as components, and likewise UnderSaps include UnderSessions as components.

A Protocol object represents a *protocol entity*: an active instance of a particular implementation of a protocol specification. Each instantiation of a given protocol in a host's protocol graph is a distinct protocol entity.

OverSap and UnderSap objects represent *Service Access Points*, or *SAPs*. A SAP is a communication service interface with an address that uniquely identifies it. The interface between one particular protocol entity and another, higher level protocol entity as identified by a given address is a SAP. The communication

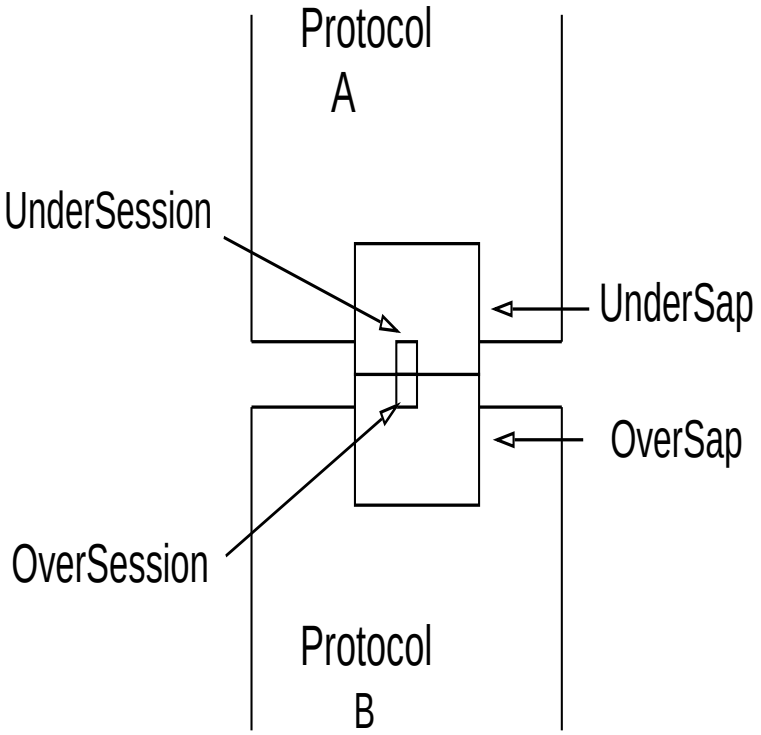

Figure 4: Sap and Session objects

service user side of a SAP is represented by a UnderSap object, which is part of the user Protocol object, while the service provider side is represented by an OverSap object, which is part of the provider Protocol (SAP objects are named from the point of view of the Protocol of which they are components; see Figure 3). Figure 4 illustrates a complete SAP consisting of a matched UnderSap-OverSap pair.

OverSession and UnderSession objects always occur in pairs; such pairs are used to represent *sessions*. A session is an endpoint of a data stream or abstract communication channel (not necessarily a connection) between two specific SAPs. Hence, there are two addresses associated with a session: the address of the SAP at "this end," and the address of the SAP at "the other end." An OverSession-UnderSession pair provides a complete two-way interface representing the local endpoint of a data stream. The two objects that combine to implement this interface each implement one direction. An UnderSession represents a one-way interface from the protocol entity that uses the channel to the protocol entity that provides the channel; a higher level protocol uses an UnderSession to pass messages and flow control information to a lower level protocol. The corresponding OverSession represents a one-way interface in the reverse direction; a lower level protocol uses an OverSession to pass messages and flow control information to a higher level protocol. Figure 5 illustrates OverSession-UnderSession pairs and data streams.

UnderSession objects have a **deliver(message)** operation (messages are delivered asynchronously, rather than being received) and a **grantSends(numberOfMessages)** operation used to convey flow control information. OverSession objects have the analogous operations **send(message)** and **grantDelivers(numberOfMessages)**.

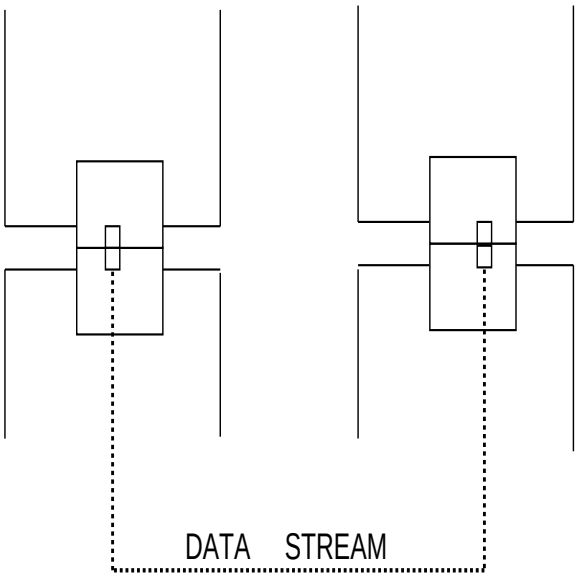

Figure 5: Sessions and Data Streams

The object operations are summarized in Table 1. For purposes of research, we have selected a minimal functional set of protocol operations; a practical system would require additional operations, such as for terminating conversations. There are two kinds of operations associated with Morpheus objects: *internal* and *external*. Internal operations are operations which each protocol must *supply*, while external operations represent the inter-protocol functions that protocols *need*. In other words, internal operations are not invoked directly by other protocols. Instead, they are invoked indirectly, by way of external operations. A Morpheus compiler could implement a particular external operation in terms of the corresponding internal operations in a variety of ways, including code generation, inheritance, and shared infrastructure routines.

EXTERNALLY INVOKED OPERATIONS	CORRESPONDING INTERNAL OPERATIONS
createProtocol(protocolClass,underSaps)	protocol.addOverSap(overSap) protocol.initProtocol(underSaps)
underSap.getLocalAddr()	overSap.getLocalAddr()
underSap.enableUpwardSessionCreate()	overSap.enableUpwardSessionCreate()
underSap.createUnderSession(addr)	*depends on the two shapes involved*
overSap.createOverSession(addr)	*depends on the two shapes involved*
underSession.send(msg)	overSession.send(msg)
overSession.deliver(msg)	underSession.deliver(msg)
overSession.grantSends(number)	underSession.grantSends(number)
underSession.grantDelivers(number)	overSession.grantDelivers(number)

Table 1: Object Operations

Notice the absence of an interface for specifying options is a deliberate aspect of the Morpheus protocol architecture. Morpheus protocols do not permit options. There are two reasons behind this constraint.

First and most important, we believe that protocol options are usually used to solve at runtime a software management problem that exists at compile time. In other words, instead of one protocol with options, there should be a distinct protocol for each value of the options. Code sharing between the variants of a proto-protocol should be managed at the source code level, not at runtime. In those cases where the appropriate variant cannot be determined until runtime, a *router* protocol (described later in this section) should be used to select the appropriate protocol on a per-session or per-message basis.

The second reason behind Morpheus' prohibition of options is to maintain the uniformity of the protocol interface, since protocol reusability depends on this. Hiding the irregularity in an ioctl interface with loosely typed arguments doesn't solve this problem since the higher level protocol still has to incorporate information about the lower level protocol's options.

Note that while the five base classes just defined must be refined by the programmer to derive specific protocols, Morpheus provides three additional classes that are not refined by the programmer: *Message*, *Map* (a kind of hash table) and *Event* (a schedulable event). These classes represent utilities frequently used by protocols. They play a role similar to library routines in other languages, but are built into the language as object classes. The programmer uses objects from these utility classes to help write the five protocol-specific classes.

Two novel aspects of Morpheus messages are worthy of note. First, conversion between the byte ordering of header fields and the (native) byte ordering of variables is transparent to the programmer. The programmer simply declares a byte order for each protocol's header, and the compiler uses the that information to generate the appropriate object code. Second, word alignment of fields in a header is maintained for more efficient access. Morpheus constrains header fields to be word aligned relative to the start of the header, and headers and data are both constrained to be an integral number of words in length.

3.3 Shapes

Morpheus constrains protocols to conform to one of three shapes: *multiplexor*, *router*, or *worker*. The purpose of this powerful constraint is to maximize the information that the Morpheus compiler can use to automatically supply code and data structures. This constraint has been carefully selected so as to avoid restricting the range of protocol functionality that can be implemented. Morpheus' shapes result from partitioning protocol functionality on the basis of "plumbing," or how many higher and lower level protocols a given protocol can interface with. Alternatively, this partition may be thought of as being determined by the kind of addressing information a protocol is responsible for processing. The relationship between addressing and plumbing will be made clearer by the descriptions of the individual shapes. The significance of this particular partition is that it provides critical information about the structure of a protocol—permitting the compiler to supply much more code than would otherwise be possible—while still supporting the full range of protocol functionality. Figure 6 schematically depicts the three protocol shapes.

Multiplexor protocols multiplex and demultiplex. They require multiplexing keys to be associated with different higher level protocols, but are ignorant of host addresses. They may use quality of service (QOS) information associated with OverSaps in performing multiplexing. A multiplexor protocol provides a variable number of OverSaps (since it may multiplex channels from many higher level protocols), but uses just one UnderSap. Multiplexors can use flow control information regarding sending messages to schedule outgoing messages, but cannot enforce flow control on delivery of messages (if needed, it must be implemented in a separate protocol).

Router protocols make runtime decisions regarding which lower level data stream (UnderSession) to use to send a message. The decision could be made on a per-message basis or a per-OverSession (higher

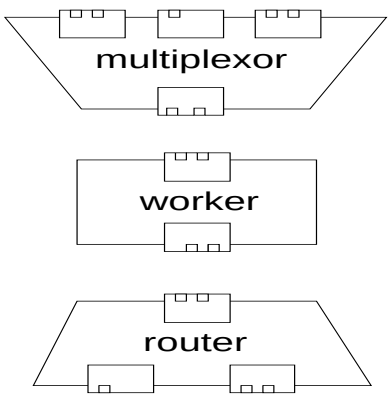

Figure 6: Three Shapes

level data stream) basis. Hence, Morpheus routers are more general than is usually suggested by the term "router" (e.g. IP), in that we include choosing between different paths within the protocol graph in a host, even if the route through the hardware network doesn't vary. A router protocol uses some fixed number of UnderSaps, and must in general be able to interpret the host address spaces implemented by each of the underlying communication services. A router protocol provides just one OverSap, and therefore never sees multiplexing keys.

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. They don't process host addresses, and they never see multiplexing keys. A worker protocol entity provides one OverSap and uses one UnderSap.

Without distinguishing protocol shapes, there is little code or state information that is common to all protocols. However, protocols of a given shape are similar enough that they can usefully inherit default behavior and state variables. For example, all multiplexors do the same thing when messages are delivered to them: demultiplex them. Hence, the **deliver** operation is completely specified for multiplexors. In contrast, little **deliver** behavior can be supplied to routers or workers because they may perform widely varying functions. For another example, each worker protocol has a single user, hence a single OverSap. A state variable representing that OverSap is automatically declared in the Protocol base class for workers. Multiplexors, which may have many OverSaps, do not have this state variable. Instead, multiplexors automatically get two state variables which are Map objects for mapping from an incoming message to the appropriate OverSap.

The Morpheus program for a protocol begins by explicitly stating the protocol's shape. This allows the Morpheus compiler to implicitly provide data structures and behavior based on the shape, thus relieving the programmer of the burden of designing and implementing them. Programmers augment the provided data structures with additional data structures, and augment or override the provided code with their own.

Note that although protocols that are *functionally equivalent* to protocols such as TCP and IP can be implemented in Morpheus, those specific protocols—as specified in their standards—cannot be implemented in Morpheus. This is because they combine the functions of more than one shape in a single protocol. For example, IP performs multiplexing, routing, and fragmentation/reassembly. In Morpheus these functions would be implemented as three distinct protocols: a multiplexor, a router, and a worker, respectively.

It is not surprising that existing protocols violate the shape constraint. One reason is the belief, refuted in this paper and [13], that efficiency requires that there be very few layers in a protocol stack. This encourages large, complex protocols that comprise multiple functionalities. A second reason is that existing protocols were designed before the current acknowledgement of the drawbacks of logical multiplexing [7, 16]; hence many existing protocols include logical multiplexing among their functions, even though they need not.

3.4 Flow Control

Controlling the flow of messages between protocol peers is a familiar problem. For example, many protocols use window- or rate-based algorithms. We call this *horizontal flow control* to distinguish it from message traffic between adjacent protocol layers, which we refer to as *vertical flow control*. Such control is needed to throttle messages from the next layer, and to propagate horizontal flow control information up or down the protocol stack to the ultimate sources of the message traffic.

Control that only responds when a message is passed to an adjacent protocol (e.g. by dropping the message or blocking the thread) is insufficient because the adjacent layer either gets no control information, or gets information only when it attempts to pass a message. While this may be adequate for some specific situations, in general it is better to propagate such information—without delay—for two reasons. The first is to throttle the ultimate sources of messages as quickly as possible in order to minimize the additional messages produced. The second is to make scheduling decisions among the messages that have already been produced. This is particularly important for satisfying quality of service requirements.

Morpheus provides an interface for propagating flow control information vertically—i.e. between adjacent layers—but does not itself enforce any flow control. In other words, Morpheus provides the mechanism, but the policy is determined by the particular protocols. Information is propagated down the protocol stack via a **underSession.grantDelivers(numberOfMessages)** operation, and up the protocol stack via a **overSession.grantSends(numberOfMessages)** operation. Messages are then passed only if credit has been granted in advance, where flow credit is expressed in units of messages. Note that Morpheus supports flow control at the granularity of Sessions (i.e. data streams), rather than Saps, and that each protocol has two, possibly identical, flow control policies—one for **send** and one for **deliver**.

This mechanism only provides a means to convey flow control information between layers; any enforcement necessary must be implemented by the protocols. One could augment Morpheus to enforce compliance by keeping track of credits and messages passed, and stopping messages when there are no credits. That approach has two major disadvantages. First, the overhead sustained is unnecessary in the common case, i.e. when the layer is trusted to comply. Second, different ways of stopping a message are appropriate in different cases. A better approach is to interpose a reusable protocol that enforces the appropriate policy.

Policies fall into three general categories, the first two of which involve little or no protocol code. The first category consists of a single policy: the bypass policy. In this case, the protocol simply relays the flow control information, and relies on its message source to comply. This makes sense 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, horizontal flow control is combined with many other functions in a single protocol, and it provides synchronization only between peer instances of that protocol. In the Morpheus protocol architecture, a horizontal flow control protocol should perform no other function, and it should propagate flow control vertically to provide horizontal flow control between peers in higher layers. It works as follows. An instance of a higher level service needs to control the flow of messages from a peer. It uses **grantDelivers** to modulate the delivery of messages from lower layers. This control propagates down to the horizontal flow control service, which translates the information into a message that propagates the information to its peer. This horizontal flow control protocol peer then uses **grantSends** to propagate the information up to the peer of the original service. Since this peer, the source of the messages, sends messages only after first receiving credit to do so, horizontal flow control is achieved.

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). Our conclusion is that 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 sends, both 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.

3.5 Inheritance

We now consider how code inherited from Morpheus shapes is integrated with a Morpheus program written for a specific protocol. The general problem of integrating superclass behavior with subclass behavior is known as the *method combination problem* [10]. The Morpheus case is much simpler than the general case because a subclass in Morpheus inherits from a single superclass, the superclass does not inherit any behavior, the superclass is never instantiated directly (it is in this sense an *abstract class*), and the programmer cannot define completely new classes.

Morpheus uses a generalization of the method combination technique used in Simula [6]. In Simula, the keyword **inner** is used in a superclass operation definition to indicate that subclass code for this operation should be executed (like a subroutine or macro) at this point in the superclass code. Unlike most other object-oriented languages, this requires that code be structured top-down—the superclass has to anticipate the ways in which it will be augmented by subclasses. This top-down structure is ideal for Morpheus since the superclasses are pre-defined by the system rather than written by the programmer.

More concretely, Morpheus does the following. In the program for a protocol the programmer writes procedures for the object operations, naming each procedure with its operation name. This code is inserted

into the base class code for the same operation. The use of procedures here is simply a syntactic convention; subclass code is combined with base class code at compile time, so there is no procedure call overhead. If a subclass doesn't need to augment the base class code for a given operation, it does not define the corresponding procedure. The procedures corresponding to some operations take parameters which differ from those of the operations. This is because the role of these parameters is to let the subclass procedure refer to context in the base class code. This is not as cluttered as it might sound because most context is implicit: the object on which the operation was invoked, and any objects of which that object is a component.

Morpheus has two features not supported by the basic mechanism just described. The first is the ability to *override* base class behavior. This permits base classes to offer *default* behavior even though that behavior might not always be desired. The second feature is the ability to intermix base class and subclass code at a finer granularity. Base class and subclass behavior do not always fall into the neat relationship required by **inner**. It is useful for some base class operations to include different blocks of subclass code, each at a different point. Extending the basic mechanism to support these features is easy because Morpheus doesn't require a general solution—the base classes are pre-defined so the instances of these features are fixed. The new code is again packaged as a procedure; the difference is that the name of the procedure is not the name of an operation. The procedure is named by a keyword corresponding to the appropriate overridable block of code or location where new code can be inserted. The keywords and their semantics are as easy to learn as the operations because, like the operations, they are few and correspond to meaningful units of behavior.

4 Examples

This section presents examples of Morpheus protocols, including one of each shape. It also gives some intuition behind the code that the Morpheus programmer does *not* have to write, that is, the boilerplate code inherited by each shape.

4.1 Worker Protocols

The code in Figure 7 is the Morpheus program for a worker protocol called Sequencer. Sequencer's function is to filter out any duplicate or out-of-order packets. Note that Sequencer does not guarantee that every message sent is delivered; if needed, that would be the function of other layers.

Figure 7: A worker protocol program

The reader should note the almost complete *absence* of any code or data structures that are *not* specific to Sequencer's function. In contrast, an implementation of Sequencer in a general purpose language (such as the C version presented in the appendix) would include data structure declarations and code for creating and assembling the component objects, connecting Sequencer to the adjacent layers, creating data streams, and pushing and popping message headers. Using Morpheus, these routine aspects of a worker protocol are all inherited. The complete Sequencer protocol can be succinctly defined because one need express only those design choices that are specific to Sequencer.

In this example, the variables **underSession** (used in **send**) and **overSession** (used in **deliver**) are examples of inherited state variables. **underSession** is an OverSession state variable, inherited from the Worker OverSession base class; and **overSession** is an UnderSession state variable, inherited from the Worker UnderSession base class. To understand what they represent and why they should be inherited, consider the nature of a worker. Since a worker does no address translation, there is a one-to-one correspondence

between the data streams the worker provides to higher level protocols and the data streams the worker uses from lower level protocols. Thus, there is a one-to-one correspondence between the worker protocol's own UnderSessions and OverSessions (in addition to the correspondence between its UnderSessions and the lower level protocol's OverSessions, and the correspondence between its OverSessions and the higher level protocol's UnderSessions). So that each OverSession knows its corresponding UnderSession to use in relaying a message, each OverSession has a state variable that refers to the corresponding UnderSession, and vice versa. In addition to these state variables, Sequencer inherits other worker state variables that are not explicit in Sequencer's Morpheus program because they are used exclusively by inherited code.

The initialization of the state variable **overSession** is an example of inherited behavior. **InitUnderSession()** is invoked when Sequencer opens a channel of the underlying communication service to initialize the UnderSession representing Sequencer's side of the interface to that channel. The inherited base class code for **initUnderSession** takes care of setting the UnderSession's **overSession** to the OverSession representing the corresponding Sequencer channel.

The keyword **LittleEndian** indicates the byte order with which the sequence number **seqNum** in the header is to be represented. The compiler uses this information to automatically generate the appropriate code for reading and writing header fields. This highlights an obvious advantage of a language designed exclusively for writing network protocols: the compiler can worry about network byte order, and the problem of word alignment is simply constrained away.

To illustrate use of the flow control interface, Figure 8 excerpts two critical routines from the Morpheus program for a worker protocol called HorizontalFlowControl. This protocol is designed for the case where the underlying service is reliable, so its message exchange conventions are very simple (e.g. no sliding window). When a HorizontalFlowControl entity receives permission (via **grantDelivers**) to pass some number of messages upward, it translates the information into a message syntax to convey it to the corresponding peer. When the peer receives the message, it converts the information into permission for a higher level protocol to pass messages downward, i.e. it consumes the message and invokes **grantSends** with the appropriate number of credits.

Figure 8: Flow control protocol fragments

4.2 Multiplexor Protocols

Multiplexor protocols provide a more dramatic example of inheritance. The code in Figure 9 is the Morpheus program for a multiplexor protocol called FCFS (First-Come-First-Serve).

Figure 9: A multiplexor protocol program

Compared to a worker protocol, more of a multiplexor is specified in the base classes because more is known about the function of a multiplexor. FCFS inherits specific algorithms and data structures for the basic multiplexing and demultiplexing tasks performed by every multiplexor. For example, a C version of this protocol included over 180 lines of actual code specific to the protocol, plus a variety of calls to library routines that are shared with other protocols.

The dimension along which multiplexors vary is the scheduling of outgoing messages. FCFS is the simplest useful multiplexor, scheduling outgoing messages on a first-come-first-serve basis. More sophisticated

multiplexors transmit messages when permitted by the flow control information conveyed via the **grantSends** operation, and transmit them in an order based on priority or quality of service (QOS) considerations.

In Morpheus, multiplexing keys identify OverSaps. 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 specifies what will eventually be an underSap for the newly created protocol. There can be only one link argument unless the protocol entity being created is a router. The form of a link argument depends on the shape of the corresponding lower level protocol. If the lower level protocol is a multiplexor, then the link argument is a tuple of the form (multiplexorEntity, key1, key2); for workers and routers, the link argument consists of an entity, without any keys. Hence multiplexing keys are neither hard-coded into the protocols nor generated by the protocols, but are specified as part of the act of simultaneously creating and composing the protocol entities. If a core graph of protocols is assumed to be always needed, its composition can be specified by a procedure invoked at boot time; for protocols added to the graph later (as in response to a demand that arises at runtime), their composition (onto an existing graph) would be specified by the software that detected the need.

4.3 Router Protocols

Router protocols vary widely in their behavior, and even in their "plumbing." For example, the relationship between OverSessions and UnderSessions depends entirely on the particular router. Consequently, relatively little of a router's behavior can be predicted by Morpheus, and routers are the shape for which the programmer must specify the most information.

The code in Figure 10 is the Morpheus program for a router protocol called Sizer 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.

Figure 10: A router protocol program

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

In general, a router protocol has to define its own host address space (more accurately, the space of ids of peer protocol entities), and know how to translate between its host address space and those implemented by the underlying communication services. This is trivial for Sizer because it is designed for the case where the two underlying communication services use the same address space, and Sizer can therefore let that host address space show through as its own.

The **getLocalAddr** interface allows any protocol entity to determine its address with respect to an underlying communication service. A router entity needs this information so it can use it to determine its address with respect to *its own* space of host addresses. Note that workers and multiplexors don't get involved with host addresses; they just relay **getLocalAddr** requests to the next lower layer until it reaches a router or a network device driver. In this respect, 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. In contrast, workers and multiplexors don't incorporate any composition information internally since they ignore host addresses, and the multiplexing keys passed to multiplexors come not from protocol code, but are specified externally when the protocols are composed.

While Morpheus does everything possible to support and encourage decomposition of protocol functionality into minimal, building-block protocols, even in the Morpheus framework there would be more complex protocols requiring more code than these examples. However, the main point that these examples illustrate is independent of the size of a protocol: the programmer can concisely express the specifics of a given protocol, and is spared the trouble of expressing the "routine" code that is common to all protocols of the corresponding shape. This effect holds whether a protocol is long or short, and we have used short examples for clarity.

5 Performance Optimizations

This section identifies some domain-specific optimizations available to a Morpheus compiler, and reports experimental results based on performing these optimizations by hand.

Morpheus protocols share a single address space, and run on a uniprocessor. Consequently, control and messages or other data are passed between layers via procedure call. We have restricted this work to the uniprocessor, single address space case in order to focus more attention on other issues.

5.1 General Strategy

Morpheus optimization techniques are based on the common patterns of protocol execution. Consider the characteristics of the **send** operation; **deliver** behaves similarly. **Send** takes a message as its argument; hence there are in effect two arguments, the message and the OverSession object. The typical **send** does some computation, accessing the object for state and other information, and using the built-in utilities to manipulate messages, hash tables, and timers; prepends a header to the message; and ends by passing the message to the next lower layer via the **send** operation of another OverSession. This is repeated as the message passes through "many" layers. Morpheus optimizes for this common case.

Morpheus optimizations are targeted primarily at minimizing per-layer costs. The main strategies are streamlining procedure linkage (since control is transferred between layers by procedure call) and factoring out computations that are repeated in multiple layers. In the best case, per-layer overhead can be reduced to two assembler jump instructions, one at the sender and one at the receiver.

Because Morpheus is intended to implement only the protocol subsystem of an operating system, the generated object code must interoperate with the operating system's object code. In particular, procedure calls in either direction between Morpheus-generated machine code and "foreign" machine code adhere to the calling conventions of the foreign code.

Morpheus' optimizations cannot be duplicated by interprocedural optimization of a general purpose language. Morpheus optimizations are subject to two major constraints not usually encountered in general purpose optimization. First, 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. Second, only the protocol subsystem is available for interprocedural optimization, not the entire operating system. Thus, protocols invoke and are invoked by foreign code, which has not been involved in the interprocedural optimizations. Even if these optimizations could be duplicated using general interprocedural optimization,

it would involve considerable interprocedural analysis at compile time. Moreover, 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, which supports separate compilation, avoids these compile time penalties. *In effect, the interprocedural analysis took place at language design time.*

Before presenting the optimization techniques, we briefly review procedure call conventions for modern RISC architectures.[1] The caller 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, which moves the return address into a designated register and transfers control to the callee. The callee then updates the stack pointer to leave enough space on the stack for local variables, temporary variables, callee saved registers, 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.

5.2 Specific Techniques

We now identify five optimization techniques employed by Morpheus. The first three are compiler optimizations in the conventional sense; the fourth is a direct consequence of using a domain-specific compiler; and the last could be performed at the source code level of a general purpose language. For clarity, the techniques are described in terms of **send**; they apply equally to **deliver**.

These optimizations target message latency exclusively. Hence their effect on performance will be most pronounced for short messages. Their impact on bandwidth performance, i.e. accessing or changing the data contained in large messages, will be relatively insignificant because the processing required in that case is linear in the size of the message instead of constant.

5.2.1 Dedicated Message Registers

Consider **send**'s message parameter, which fits in a register because it is implemented as a pointer. If **send** calls any procedures (other than those which take the message as an argument, in the same order in the argument list), the message has to be saved so that another argument can be passed in the argument-passing registers. Ultimately it must be restored to its original argument-passing register to be passed to the next layer's **send**. 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 what's more, that register need not be freed across subsequent calls to either the next layer's **send** or any other procedures.

The part of the message used most heavily 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

[1] We have decided to not consider register windows in this work, as we expect them to play a diminishing role in future machine architectures.

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 **send** to obtain a per-layer cost. The message and header registers can potentially be reallocated within a **send** if registers are in sufficiently short supply or if a second message must be passed, but this case is the exception.

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

This optimization could be described as a second procedure calling convention that coexists with a primary calling convention. Interprocedural optimization has the potential to customize the calling interface between a pair of procedures, but interprocedural optimization can't be applied to this case because it isn't determined until runtime which layers are adjacent to each other. Furthermore, interprocedural optimization entails a compile time cost which this Morpheus optimization does not. It is as though Morpheus performed the interprocedural analysis at language design time.

5.2.2 Short-Circuit Return

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

This short-circuit return optimization in itself saves relatively little—a single jump assembler instruction per layer on the MIPS processor. However, it contributes to another, conventional optimization that *is* significant. If there are no procedure calls in a **send** 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 and performing the optimizations described below, a significant number of **send** operations qualify as having no procedure calls. This occurs frequently since the typical Morpheus protocol is relatively simple.

To summarize, this optimization pays off for Morpheus for several reasons. First, many, perhaps most layers simply return after invoking the subsequent layer, so this optimization would apply. Second, because layers are simple, many of them will have no procedure calls other than the call to the next layer, and will therefore be able to delete at least five instructions. Third, again because layers are simple, deleting one or a few instructions from a layer can be significant. Finally, because there are many layers, the per-layer savings is repeated many times in a protocol stack.

Why isn't this optimization implemented for general purpose languages? Because the benefit for general purpose *programs* is so small on average that it is not worth implementing. In contrast, the Morpheus **send** and **receive** operations present a more restricted 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 the utility operations. Consider a **send** in which the sole procedure call is in some inlined utility code in a branch that is known to be infrequently taken. Instructions to manage the return address and stack pointer registers—i.e., a "lazy stack"—are inserted just in that infrequent branch, so that they are executed

only if necessary.

5.2.3 Procedure Cloning

Send almost always accesses instance variables in its OverSession objects since these hold connection state information and other information such as the appropriate lower level OverSession object. It also frequently accesses instance variables of the Sap and Protocol objects to which the Session object belongs. Morpheus optimizes for this by generating a customized version of the **send** object code for each OverSession. At compile time, Morpheus generates a template for each protocol's **send**. 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 state variables that are known to be constant. Most inherited state variables 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. A user-declared state variable can be flagged as a constant by a keyword. Chains of indirect pointers through memory are collapsed; for example, the address of the next layer's **send** 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 are different for each clone, hence they can't be hardwired into an uncloned procedure). This reduces the number of instructions executed for each clone. More importantly, it eliminates the memory accesses—disproportionately costly on a RISC machine—that would otherwise be necessary to read these constants.

This technique is a variation on *procedure cloning* [4]. 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 [8]. 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 cannot 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 [14] achieves exceptional performance using a similar technique. However, in contrast to Synthesis, which generates customized kernel code, Morpheus generates customized 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. While this does occur at runtime, it is part of communication channel creation—not in the time-critical **send** path. The space cost is an extra copy of the **send** 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 **send** 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, and because some of the context-state is hardwired into the code. The increase in code space can be bounded by simply ceasing cloning once a code space threshold has been reached, as proposed in [8]. This would require keeping one uncloned version of each **send** procedure to operate on any OverSessions that weren't allocated their own clones.

Increased object code size due to inlining (not cloning) seems to have little effect on caching and virtual memory. [5] found no obvious evidence of either thrashing or instruction cache overflow, 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.

5.2.4 Language Constructs for Frequent Tasks

Morpheus provides language constructs for frequent protocol tasks such as manipulating messages, mapping identifiers, and setting timers. 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), language constructs offer 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.

5.2.5 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.

5.3 Experimental Results

To study the impact of Morpheus' optimizations in the absence of a compiler, we 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; then finally applying the optimizations by hand at the assembler language level. We then performed two experiments to quantify the effect of Morpheus' optimization strategy: counting instructions and measuring end-to-end latency.

5.3.1 Instruction Counts

The effect of a given optimization depends on both the particular procedure and the other optimizations present. Hence, we have selected a particular protocol to use as an example, and report the effects as each optimization is applied to it in turn. The protocol is Sequencer, which was presented in the previous section. We focus on Sequencer's **send** operation. When Sequencer's **send** 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 **send**.

The results of the optimizations are summarized in Table 2. 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.

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

Table 2: Instruction Counts

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 **send** (Sequencer has no 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 **send** eliminates another seven instructions. Several pointer indirections are short-circuited, and one less parameter is passed to the next **send** (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 **send** 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 **send** consists of seven instructions: one to increment the header pointer, five to do "the real work", 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 gap between processor speed and memory speed can only be expected to widen in the future, making memory accesses an even more dominating factor in performance. 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.

5.3.2 Timing Measurements

We 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' purported performance advantages would result in measurably high performance. The *x*-kernel was used as the standard for comparison because we 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 [11]. 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.

We measured the end-to-end latency of our two versions of UDP—the time it takes one message to be sent and received, independent of all other protocols. The measurement was taken by sending and receiving ten million, 1-word messages, and dividing the elapsed time by ten million. In this experiment, latency was independent of message size because the optional UDP checksum was not performed, neither system copies a message to pass it between layers, and messages were not actually transmitted over a network. 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 our experiment because source and destination shared the same processor. In this particular case, the additional time is relatively insignificant because it only involves three procedure returns. This was not a factor for the Morpheus time because Morpheus' 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. We argue that this is a legitimate advantage, attributable to the "many, simple 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.

5.4 Discussion

There are two conclusions to draw from these experimental results. The first is that by using optimization techniques available to a special purpose language, a Morpheus compiler can generate very fast object code.

The second conclusion is that per-layer overhead in Morpheus is negligible. By "per-layer overhead" we mean the additional end-to-end latency of a protocol that is due to implementing it as a distinct protocol instead of incorporating it in another protocol. Sequencer's overhead is four instructions; two from **send** and two from **deliver**. Protocols more complex than Sequencer entail more overhead (because they need stacks and temporary registers and so on), but the overhead at each end is still less than a procedure call.

An argument could be made that combining multiple functions in a single protocol layer still results in less overhead. It is true that the relative overhead—the ratio of overhead to "real work"—generally decreases as the functionality is squeezed into fewer layers. However, performance is not the sole issue. Combining functions in a single layer buys performance at the expense of modularity and its advantages in developing, verifying, implementing, and maintaining complex functionality. Morpheus not only supports modularity at the right level of granularity—indivisible protocol functions—but also offers benefits beyond conventional static modularity by supporting runtime composition of modules. All these benefits come at an end-to-end latency cost of less than two procedure calls per layer.

Latency of the protocol stack is not the only performance issue for network software; maximizing end-to-end throughput is also a pressing problem. The optimizations presented here focus on latency and do not make a significant contribution to throughput. The work described in this paper is just a part of a larger effort to address both latency and bandwidth in the context of a more powerful programming environment. Our approach to bandwidth optimization is described in [1]. Although end-to-end throughput is generally considered to be the more important performance issue in high-speed networking, minimizing latency is still a critical goal. Considerable effort has been expended optimizing the latency of TCP/IP [3], obtaining both a significant performance benefit, and evidence bearing on the claim that TCP/IP latency need not be a performance bottleneck. The optimization techniques we introduce are not specific to any one protocol stack such as TCP/IP; hence we obtain both a significant performance benefit *for protocols of any functionality*, and evidence bearing on the claim that *protocol latency in general* need not be a performance bottleneck. This paper emphasizes per-layer latency in particular to support our thesis that highly layered architectures need not entail any significant latency penalty over architectures with few layers.

6 Concluding Remarks

Morpheus is a special-purpose programming language that facilitates the implementation of efficient communication protocols. In the context of implementing network software, our objective is to explore the design space that lies between implementing protocols by hand in the host operating system, and automatically generating network software from formal specifications. The key to Morpheus is that it constrains the protocol programmer. Morpheus' constraints enforce a good design discipline, relieve the programmer of many low-level design and implementation tasks, and admit optimizations for high performance. A powerful constraint unique to Morpheus is that of protocol shape.

References

[1] M. B. Abbott and L. L. Peterson. Automated integration of communication protocol layers. Technical Report 92-25, Department of Computer Science, University of Arizona, Dec. 1992.

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

[3] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

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

[5] K. D. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 91.

[6] O.-J. Dahl and K. Nygaard. Simula—an Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, Sept. 1966.

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

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

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

[10] B. L. Horn. An introduction to object oriented programming, inheritance and method combination. Technical Report CMU-CS-87-127, Computer Science Department, Carnegie Mellon University, Jan. 1988.

[11] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[12] G. Kane. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[13] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2), May 1992.

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

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

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

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

[18] R. W. Watson and S. A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 5(2):97–120, May 1987.