

# Protocol Portability through Module Encapsulation\*

*Bobby Krupczak*

*Ken Calvert*

*Mostafa Ammar*

**GIT-CC-96-12**

April 12, 1996

## **Abstract**

Because protocol software is difficult and expensive to implement and test, it is often ported between systems, instead of being rewritten from scratch. Unfortunately, porting protocol software can be nearly as difficult as from-scratch development, due to inherent differences in subsystem design and services provided. Thus, protocol subsystems can have a profound effect on the portability of a protocol implementation. In this paper, we propose an approach permitting the incorporation of new protocols into a subsystem other than their “native” one, without the drawbacks or expense of porting and original development. Our approach is based on protocol module encapsulation, which allows unmodified protocol code developed for one protocol subsystem to be used within another. We relate our experiences designing, implementing, and measuring the performance of our protocol encapsulation modules, using an AppleTalk protocol stack as a baseline.

College of Computing  
Georgia Institute of Technology  
Atlanta, GA 30332-0280  
{rdk,ammar,calvert}@cc.gatech.edu  
404.894.1404  
404.894.0272 (fax)

---

\*This research is supported by a grant from the National Science Foundation (NCR-9305115)

# 1 Introduction

The environment within which it is developed can have a profound effect on a protocol’s implementation and its portability. Indeed, protocol code written for one system may be unusable in another. Because protocol software is complex and difficult to implement and test, it is usually developed within protocol *subsystems* (e.g. Streams [24], BSD [16], or the *x*-Kernel [19]). When a new protocol needs to be added to a system, an existing implementation is often ported, instead of developing a new one from scratch. In fact, empirical evidence indicates that many of today’s Internet protocol implementations are still derived from the original BSD implementation. Unfortunately, porting protocol software can itself be as difficult (or more so) than original development due to inherent differences in protocol subsystem design, the services they provide, and the structure they impose. This difficulty *protocol portability problem* has prevented the quick incorporation of new protocols and, in some cases, limited interoperability. The continuous introduction of new operating systems and accompanying protocol subsystems only exacerbates the problem as protocol programmers are forced to port or re-implement existing protocols.

Our previous work [14] also examined the difficulties encountered when porting protocol implementations. That work, which focused on environments in which multiple protocol subsystems are supported, introduced a subsystem adapter module which allows protocol implementations residing in different subsystems to be combined into a single protocol graph (termed a multi-subsystem protocol graph). Consequently, instead of being ported, protocol implementations can remain in their native subsystem. In this paper, we address the difficulty and expense of protocol porting and original development by proposing a different approach that avoids both. This new approach is oriented towards systems that possess only a single protocol subsystem; it allows protocol source code originally developed for one subsystem to be imported and used without modification in another subsystem through module encapsulation. We describe our experiences designing, implementing, and measuring our module encapsulation technique and the resulting protocol graphs which we construct. The key difference (between this and our previous work) is that we focus here on environments in which only a single protocol subsystem is supported. Consequently, protocols cannot remain in their native subsystem; instead, their implementations must be imported to the native subsystem.

The rest of the paper is organized as follows. First we provide background information and discuss related work in the next section. Next, we provide an overview of protocol encapsulation in Section 3 followed by a discussion of its application and implementation for the BSD and Streams subsystems in Section 4. Section 5 discusses its performance while Section 6 compares the encapsulation approach to others. Section 7 concludes the paper.

## 2 Background

### 2.1 Definitions and Terminology

For the purposes of this paper, a *protocol* is a software module that corresponds to an implementation of a traditional, monolithic protocol specification like TCP or smaller protocol “functions” that have become popular in the current literature. Protocols execute within the context of a *protocol subsystem*, which

organizes operating system resources like buffers and timers in a manner intended to ease the burden of protocol development. Protocols are arranged in a graph structure (commonly referred to as a *protocol graph*) representing how those protocols are combined to provide communication services. Nodes in this graph represent protocol implementations and edges represent their interconnection; protocol graphs are oriented such that protocols are connected to from above and below. Users are situated at the top while the “network” or physical medium is situated at the bottom. *Input* and *read* refer to data or control flowing upwards through the protocol graph while *output* and *write* refer to data or control flowing downward, to the network.

Protocols interface with the subsystem and other protocols to provide communication services to other protocols or users; protocols may manipulate data, add or remove protocol headers. Protocols are invoked both by the subsystem and by other protocols. We refer to the subsystem into which protocol code is placed as the *host* subsystem while the subsystem from which it originated is referred to as the *target*. The act of *porting* protocol implementations involves isolating and translating that portion of the protocol implementation dependent on the target subsystem. Finally, we define *importation* as the overall process which takes protocol code from the target subsystem and embeds it within the host subsystem.

## 2.2 Related Work

A considerable amount of work has examined protocol implementation problems, protocol subsystems, and their respective performance. However, little research has directly examined the protocol portability problem. In this section we review relevant work and analyze how it relates to our efforts aimed at reducing or eliminating the protocol portability problem.

Clark *et al* [7, 8] address protocol interoperability by proposing an architecture in which protocols are mixed and matched until two communicating entities support a common protocol graph. They advocate that systems support as many protocols as possible but do not directly address the protocol portability problem. A substantial body of work, in the area of protocol conversion, has also focused on achieving protocol interoperability but has done so by addressing their visible output (e.g. headers and message formats) [10, 15, 5]. However, protocol conversion abstracts away from a protocol’s implementation and its subsystem.

Others [19, 24, 16, 20, 13, 21, 6, 28, 1] have focused on making the protocol subsystem “better” in terms of performance and ease of programming. However, protocol code portability is not addressed; further, by introducing new subsystems, they only add to the protocol portability problem. Indeed, a new release of the *x*-Kernel (version 3.3) introduces incompatibilities with previous versions which necessitate the porting of protocol software between them.

In Base [9], the authors directly address the protocol portability problem by shifting the focus away from the protocol implementation and to the protocol subsystem. By defining a single protocol subsystem (and an accompanying protocol model and set of interfaces) and implementing it across several different operating systems, protocol portability barriers are reduced. However, it is unclear just how expensive it is to port the subsystem to each new operating system and how much performance is lost in that subsystem’s generalization. Further, they do not address how existing protocol software can be incorporated into this

new subsystem without the expense of porting or re-implementation.

Various standards-based approaches (e.g POSIX [11, 12] and X/Open [29]) have addressed application-level portability across various operating systems. These standards focus on the definition of generic operating system and protocol-independent application programmer interfaces. In effect, these standards attempt to impose a common programming model and set of interfaces on application programmers. Portability is achieved because differences in the underlying systems are removed. These efforts, however, do not address portability in system or protocol software and, instead, focus only on applications.

The software engineering community has examined similar problems. For example, “wrapper” technology has been proposed as a means of retrofitting older, existing software so that it is usable in new programming environments. Schmidt [25] describes the construction of C++ wrappers that hide the idiosyncrasies of UNIX IPC programming behind an object-oriented front. However, the underlying protocol implementations, or the subsystems in which they reside, are not considered. Module interconnection languages [23, 2, 22] have addressed module reuse through the application of formal methods to software interfaces and their specifications. They, however, have not examined system software or protocol implementations nor their performance implication. Lastly, there has been extensive work in the domain analysis area [3] but it tends to be very domain-specific. We are unaware of any analysis of the protocol and protocol subsystem domain. To some degree, our current and previous work is the beginnings of such an analysis.

### 3 Protocol Encapsulation Modules

In this section, we describe a general approach that allows protocol programmers to take protocol code originally developed for one subsystem (the *target* subsystem) and use it unmodified in another subsystem (*host* subsystem). Because we wish to avoid modifying the original protocol implementation (as well as the host subsystem), we must provide for or emulate the “natural” environment in which the protocol originally operated. We do so by incorporating the original, unmodified protocol implementation (that is, its source code) within a new module — a *protocol encapsulation* module. We first describe the assumptions we make and then present the overall functionality of protocol encapsulation modules.

Our approach involves a tradeoff between network performance and the cost of porting protocol implementations: it is difficult to perform this type of conversion without incurring some performance penalty. The underlying premise is that the ease of making a new protocol available in a new subsystem will offset any (modest) performance hit, at least until a native implementation becomes available. We do not propose the approach as a replacement for porting or from-scratch development; rather, in an era of proliferating protocols, it is a technique for reducing some of the logistical barriers to the success of new services, which often depends on rapid deployment.<sup>1</sup>

Let us now consider the functionality required for our approach. Figure 1 depicts the design of a generic protocol encapsulation module, which essentially emulates a “native” protocol to the host subsystem, while

---

<sup>1</sup>Previous work with multi-subsystem protocol architectures [14] examined the interaction between a protocol implementation and subsystems and their effect on protocol code portability; that work provides the basis for making the claim that porting protocol implementations is expensive, difficult, and time-consuming.

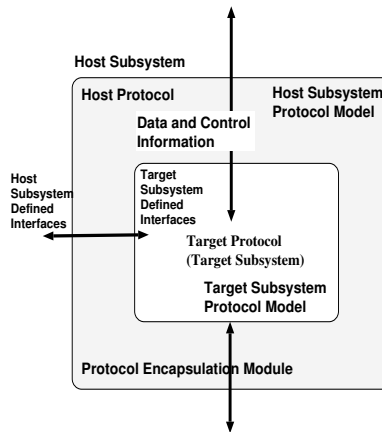


Figure 1: Protocol Encapsulation Design

simultaneously emulating the target subsystem to the imported protocol. This involves several tasks: First, the encapsulation module translates services offered by the host subsystem into a form compatible with those offered by the target subsystem (and expected by the protocol itself). Second, it augments the host's subsystem to provide those services present in the target subsystem but not the host. This may involve using other facilities of the host operating system. Third, it translates the interface between protocols themselves, i.e. other modules above or below the imported protocol in the stack. One thing a protocol encapsulation module does *not* do, however, is produce or consume headers: the encapsulation code works transparently, and ideally does not affect the “bits on the wire” at all.

In general, the functionality of a protocol encapsulation module will be specific to its particular host and target subsystems. For a given host-target pair, however, most of the functionality of an encapsulation module can be expected to carry over from protocol to protocol, with a small amount of specialization necessary to deal with differences such as handling of options and argument passing.<sup>2</sup>

Among the most important determinants of a protocol encapsulation module's functionality are the different *protocol models* used by the target and host subsystems. The protocol model dictates how a protocol interfaces to both the subsystem and other protocols. Thus a protocol encapsulation module must translate between the syntactic and semantic aspects of the target and host subsystems' interfaces. Examples of the functions of such interfaces are opening and closing a session, and sending and receiving data. The protocol model also defines the manner in which protocol entities like messages, layers, connections, and buffers are bound to the underlying unit of scheduling, called a subsystem's *process architecture* [26]. Consequently, protocol encapsulation modules must also translate between the respective process architectures. Example process architectures include horizontal (scheduling protocol layers) and vertical (scheduling connections or messages). The protocol model also defines *protocol graph connectivity* options, which include support for layering, graph construction, and graph alteration. For example, some subsystems support the dynamic

<sup>2</sup>Because all encapsulation modules for a given pair of target and host subsystems perform similar operations and share a common design, their implementation requirements would seem to present an excellent opportunity for object-oriented development techniques supporting inheritance and specialization. Consideration of such techniques, however, is beyond the scope of this paper.

construction and destruction of protocol graphs while others fix the protocol graph at system load time.

To reduce the complexity of protocol development, subsystems often provide generic, protocol-independent support services like buffers, routing table support, and timers. Because these services can be an integral component of a protocol implementation, encapsulation modules must also provide for those support services present in the target subsystem but not offered by the host subsystem. For example, the BSD subsystem supports periodic timeouts for scheduling future protocol processing but some host subsystems may only provide the ability to schedule threads for such tasks. In this case, the host subsystem must provide timeouts and timers using its own, native thread services.

Protocol implementations often build on the services of other protocols in order to provide their own; these protocol-protocol service requests must also be translated and forwarded by the encapsulation code. For example, protocols may exchange control information with one another in the form of lower and higher layer addresses, queue sizes, session state, etc. The host and target subsystems may use different encodings or conventions for this information, and these differences must be resolved.

Note that in this paper, we do not consider the problems and difficulties associated with programming language differences and porting code between them. Instead, we assume that the host and target subsystems as well as the protocol implementation in question are coded within the same programming language. In most systems today, this assumption is not unreasonable. For example, most UNIX variants, their accompanying protocol subsystems, and protocol implementations are all coded in the C programming language.

## 4 Importing BSD Protocols into Streams

Following the motivation and design presented so far, we next present the application of our approach to the importation of BSD-implemented protocols into the Streams subsystem. We first motivate our choice of the BSD and Streams subsystems and then discuss our approach's application to the environment we chose.

Our work was conducted in an environment consisting of Solaris 2.4, the Streams and BSD subsystems, and an AppleTalk [27] implementation. We chose this environment for several reasons. First, the AppleTalk suite of protocols provide a set of services similar to the Internet family and is in widespread use. Second, the AppleTalk protocol suite is not biased towards or against the BSD and Streams subsystems. We chose the BSD and Streams subsystems because they are widely deployed and are used for developing commercial and research protocols. We chose the Solaris 2.x operating environment (over SunOS 4.1.x) since the BSD subsystem does not exist within it. We also chose the combination of BSD, Streams and AppleTalk because we used them in a prior protocol-subsystem case study [14] and could build on its results. Finally, this environment solves a real problem, namely the need to incorporate BSD-coded protocols within the Streams subsystem.

Although we used AppleTalk as the basis for our implementation and testing, our design is based on the analysis of that AppleTalk implementation as well as the native BSD/SunOS Internet implementation. We analyzed the source code for each family of protocols and incorporated their subsystem requirements into the design of our BSD protocol encapsulation module. Our encapsulation module design, depicted in Figure 2, permits the importation of unmodified BSD-coded protocols into the Streams subsystem. We next

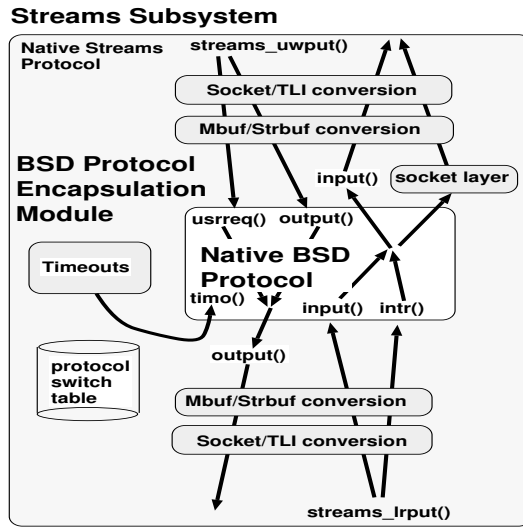


Figure 2: BSD Encapsulation Module (within Streams)

discuss the design and operation of this encapsulation module. We divide the discussion into those aspects concerned with the host subsystem (Streams) and those aspects concerned with the target subsystem (BSD).

## 4.1 Host Subsystem Translation

The BSD protocol encapsulation module must convert between the differing protocol models, translate data and control information flowing through it, and accommodate the different process architectures.

### 4.1.1 Protocol Model Differences

**Data transfer.** The BSD protocol subsystem defines a set of protocol methods or functions that are invoked by the subsystem and by other BSD protocols. However, these protocol methods differ from those defined by the Streams subsystem. The BSD subsystem defines input and output processing methods roughly corresponding to the reception and transmission of packets while Streams defines methods for reading and writing. The BSD protocol encapsulation module must support the conversion between read and input and write and output methods. When data arrives at the BSD protocol encapsulation module, it is in the format suitable for a Streams protocol. The BSD protocol encapsulation module then converts the Streams message into the appropriate format and invokes the BSD protocol's input, output, user-request, or interrupt method where appropriate. Because BSD protocols can invoke each other directly (e.g. calling another protocol's output or input function), the BSD encapsulation module must also capture and convert these direct invocations. The BSD encapsulation module does so by defining output and input functions for all higher and lower layer protocols that the target protocol may invoke. When these functions are invoked, they convert from the BSD to Streams format.

**Timeout Processing.** The BSD subsystem defines methods for fast and slow timeout invocation that invoked approximately every 250 and 500 milliseconds respectively. Protocols may utilize one or both of these timeouts to perform periodic processing or “housekeeping” chores such as re-transmitting packets, clearing internal state information, or probing the status of a peer protocol. Service queues are the method for deferred processing in the Streams subsystem. However, service queues and their accompanying service methods are scheduled non-deterministically and are, therefore, a poor match for the more the fast (250 millisecond) and slow (500 millisecond) timeout methods. Further, although matching service queues to fast and slow timeout methods may be syntactically possible, it may also cause the BSD protocol to function incorrectly. For example, the non-deterministic scheduling of service queues may cause a BSD transaction protocol to re-transmit packets too fast or too slow. This re-transmission change may cause the protocol to fail to conform to the protocol specification and could lead to interoperability problems. Because of this mismatch, the BSD protocol encapsulation module utilizes the UNIX kernel’s own generic timeout facility to provide BSD fast and slow timeouts. The BSD protocol encapsulation module could have utilized Solaris kernel threads to support timeouts, but this feature (kernel threads) may be absent in other UNIX kernels and was therefore not used.

**Initialization.** The BSD protocol subsystem defines an initialization method that is invoked when the system is first loading. When invoked, BSD protocols typically initialize internal state tables and perform control operations to other protocols (e.g. determining a lower-layer address). While Streams does not define an initialization method, the Solaris 2.x Device Driver and Device Kernel Interfaces (DDI/DKI) [17, 18] do. Therefore, it is straightforward to map between them. Because BSD protocols can also refer to each other via the BSD subsystem’s protocol switch table, the BSD protocol encapsulation module also defines and maintains one as well. The BSD subsystem protocol switch table is a data structure that lists the protocols and their types that are currently contained within it. Defining and maintaining a protocol switch table permits the encapsulation code to abstract from the particular target protocol; this abstraction fosters reuse by reducing the direct dependencies between encapsulation module and the target protocol.

**Protocol Graph.** BSD protocols are composed into a protocol graph at the time the kernel is linked. However, Streams protocol graphs are built and configured at run-time. The BSD protocol encapsulation module must resolve these differences when necessary. For many protocols, this difference is negligible and is never an issue. However, for some protocols, this difference can be important. For example, upon initialization, network layer protocols (e.g. DDP or IP) may query lower-layer network-interface protocols in order to learn their MAC addresses. In the BSD subsystem, this task can be safely accomplished at initialization time since the protocol graph was pre-constructed. However, in the Streams subsystem, the protocol graph is unknown at the time each protocol is initialized, so a direct mapping between Streams and BSD initialization methods may not work. In these cases, the BSD protocol encapsulation module must defer invoking the target protocol’s initialization method until construction of the Streams protocol graph has been completed.

### 4.1.2 Data and Control Flow

The protocol subsystem defines the syntax and semantics of data and control information exchanged between protocols and the subsystem. However, the format used by BSD differs from that used by Streams. If an unmodified BSD protocol is to function properly within Streams, then the encapsulation code must translate all data and control information as it passes to and from the Streams subsystem as well as to and from other Streams protocols. In the following discussion, we differentiate between data and control information for two reasons. First, both the Streams and BSD subsystems (as well as the *x*-Kernel) do so. Second, the interfaces for exchanging control and data information differ so it is logical to treat them separately as well.

The BSD subsystem defines two separate and incompatible (with each other and with Streams) interfaces for the exchange of control information. The first, used between BSD protocols, utilizes the control input and output methods. The second, for control information exchange between the socket layer (or users) and protocols, utilizes the user-request method. The Streams subsystem, on the other hand, uses a single interface for control information exchange, but differentiates between control messages between protocols and control information originating or terminating with user processes. This differentiation is based on the Streams message type. Consequently, the BSD protocol encapsulation module must recognize and differentiate socket-protocol and protocol-protocol control exchanges and translate them appropriately. Likewise, the BSD protocol encapsulation module must capture and convert any control invocations (of other BSD protocols) that the target protocol may make. It does so by defining control output and input functions for all higher and lower protocols that the target protocol may invoke. When these functions are invoked, they convert the control operation format.

While protocol-protocol and protocol-user control operations are fairly straightforward to translate, converting protocol-subsystem operations is not so simple. For example, the Streams subsystem defines several message types (e.g. flush, start, stop, and hangup) used by the subsystem to implement inter-module flow control and to signal interruptions in processing. These Streams control operations have no equivalent in the BSD subsystem. Consequently, our BSD protocol encapsulation module is faced with three options: ignore these control operations when they may arise (and risk blocking the system or putting it in an undefined state), return an error without performing the option, or return success without performing the operation. Our BSD protocol encapsulation module responds to flush operations by returning success. We take the optimistic approach and ignore all others. Fortunately, these operations are either not applicable (e.g. hangup) or seldom used (e.g. start and stop).

In the Streams subsystem, all data and control operations are formatted as messages passed between protocols. These messages typically conform to “standards” or conventions which govern their syntax and semantics. The Transport Layer Interface (TLI) [4] is one commonly used convention. However, the BSD subsystem defines a set of function call arguments and operands that are passed to a protocol’s user-request method. Consequently, the BSD protocol encapsulation module must translate between the different TLI messages and their corresponding BSD argument and operand formats. Fortunately, TLI appears to be patterned after the BSD socket API and, therefore, no significant incompatibilities exist.

### 4.1.3 Process Architecture Differences

The BSD and Streams subsystems differ slightly in their process architectures. The BSD subsystem is strictly vertical in nature with a thread, process, or interrupt escorting a packet through the protocol graph in a series of function calls. Streams, however, utilizes a combination of horizontal and vertical process architectures. This difference does not present major problems but is less than ideal. For example, error and status codes are returned as messages in Streams and their correlation to specific data messages is weak. However, in BSD, return codes are immediate and well-correlated to a particular data input or output operation. Consequently, when data originates in Streams and is passed to an imported BSD protocol, error and status feedback is immediate. When data originates in an imported BSD protocol and then is passed into Streams, the BSD protocol encapsulation module can only return immediate feedback if an error occurs during that translation process but cannot return feedback from other, downstream or upstream Streams protocols. In this case, the BSD protocol encapsulation module returns success even though the operation may fail as it progresses through the rest of the protocol graph.

## 4.2 Target Subsystem Service Accommodation

The BSD protocol encapsulation module must also provide those facilities used by the imported protocol but not present in Streams. Two very important services provided by the BSD subsystem, *mbufs* and the socket layer, are needed in order for BSD protocols to function properly within Streams.

### 4.2.1 *mbuf* Support

Although the Streams subsystem provides its own buffer support (Streams *mblks*), our BSD protocol encapsulation module cannot simply translate *mbuf* calls to their equivalent Streams *mblk* calls for several reasons. First, both *mbufs* and Streams *mblks* do not support a strict interface that hides the protocol programmer from their internal implementations. Second, protocols often directly access and manipulate their internal structures instead of using more well-defined function-call or macro interfaces. Lastly, because buffer manipulation is so pervasive in protocol implementations, supporting *mbufs* through translation could lead to poorer performance. Had *mbufs* had a more strict buffer interface (like that in the *x*-Kernel in which the protocol programmer is hidden from the details of the buffer implementation and only permitted to access them through a well-defined interface), perhaps *mbufs* could have been efficiently translated instead.

Because we desired a minimal performance impact, we chose to provide BSD *mbuf* support through the encapsulation of the underlying Streams buffer facility. This encapsulation provided several advantages. First, we could build upon the already existing Streams *mblk* code. Second, as the underlying Streams *mblk* implementation's performance improved through tuning, so would our encapsulated *mbufs*. Third, our encapsulating scheme could avoid data copying when converting between *mbufs* and Streams *mblks*. When converting from *mbufs* and Streams *mblks*, only pointers need be adjusted. When converting from Streams *mblks* to *mbufs*, only an additional (hopefully cached) and empty Streams *mblk* need be allocated and set to point to the original Streams *mblk*'s data. Once allocated, an encapsulated BSD *mbuf* can be manipulated without any conversion or translation. Although the particular version of *mbufs* that we implemented are

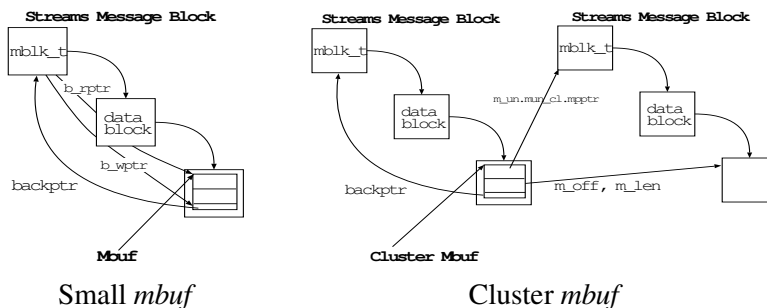


Figure 3: BSD *mbuf* encapsulation within Streams *mblks*

those supported in SunOS 4.1.x<sup>3</sup>, our encapsulation method can apply to any flavor of *mbufs*. Figure 3 depicts how both small and cluster *mbufs* are encapsulated within native Streams *mblks*. We measure the performance impact of our *mbuf* encapsulation scheme in a later section.

#### 4.2.2 Socket Layer Encapsulation

Within the BSD subsystem, separate interfaces are defined for protocol-user and protocol-protocol interaction. The socket layer, and associated socket structure and system calls, provides a protocol-independent abstraction for protocol-user communication in BSD-based operating systems. The socket layer, analogous to a Streams head, facilitates communication between user processes and protocols by providing a temporary hold place for data destined for or received from the network. Because BSD protocols communicate with user processes via the socket layer and because BSD protocols directly access socket structures, our BSD protocol encapsulation module must also support the socket layer abstraction in order for BSD-coded protocols to operate unmodified within the Streams subsystem.

The BSD protocol encapsulation layer, however, need not support the entire socket layer since it is conveniently divided into an upper and lower half. The upper half of the socket layer interfaces primarily with socket-layer system calls while the lower half interfaces with protocols. Since we are encapsulating BSD protocols within the Streams subsystem, we have no need to support the BSD sockets API<sup>4</sup>. The encapsulated socket layer need only support the socket structure and a few function calls to manipulate it.

When a packet destined for a user process is received, several things happen. First, the BSD protocol appends the packet to the end of the queue of data stored within the socket structure. Next, the BSD protocol issues a function call that awakens any user processes that may have been sleeping on that socket while awaiting the arrival of data. When the wakeup call is issued by the BSD protocol, our encapsulated socket layer then removes the data from the socket, converts it from BSD to TLI, and sends the data upstream.

<sup>3</sup>The various flavors of BSD have differing (and often incompatible) *mbuf* structures

<sup>4</sup>Actually, on many non-BSD systems, the sockets API is supported as a user library which is then translated to the Streams API.

## 5 Protocol Encapsulation Performance

In this section we present the results of performance measurements on the various protocol graphs and protocol encapsulation modules constructed (see Figure 4). During the course of our work, we have developed two protocol encapsulation modules and have constructed three different protocol graphs that combine both native and imported protocols. We measure the performance of several aspects of our approach: the overall throughput obtained by the various protocol graphs, the protocol processing time incurred by the individual protocols, our encapsulation modules, and their sub-components. As a baseline, we compare (where appropriate) our results from those obtained in earlier work [14].

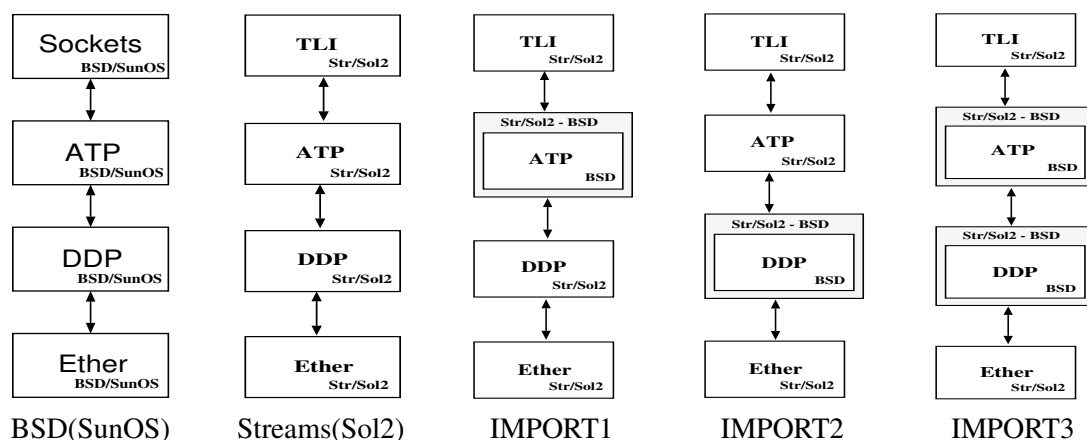


Figure 4: Native and Hybrid Protocol Graphs

All our measurements were taken on a Sun SPARCstation-LX running in single-user mode and were averaged over multiple samples within a 95% confidence interval of at most 10% of measured values. For Solaris measurements, denoted using *Sol2*, we used Solaris 2.4; for SunOS measurements, denoted *SunOS*, we used SunOS 4.1.3\_U1. Further, because all our protocol implementations are derived from a common source, their performance differences reflect only those portions of a protocol implementation dependent on the subsystem.

We label our graphs and components using the following convention. The particular subsystem in use is denoted first followed by the particular operating system in parenthesis. For example, *Str(Sol2)* indicates that the Streams subsystem inside the Solaris 2 operating system is under consideration while *BSD(StrSol2)* denotes that a BSD protocol encapsulated within Streams within the Solaris 2 operating system. We refer to protocol graphs containing some native and some imported protocols implementations as “hybrids”.

We continue to use this hardware and software environment for two reasons. First, because previous work compared subsystem performance issues using this environment, and because we wish to build on that work, keeping a consistent hardware and software platform fosters better, more accurate comparisons. Second, because we are interested in the relative performance impact that our protocol encapsulation approach has

on protocol processing performance, rather than the absolute throughput obtainable by our protocol graphs, the relative speed of the underlying system is less important.

## 5.1 Overall Throughput

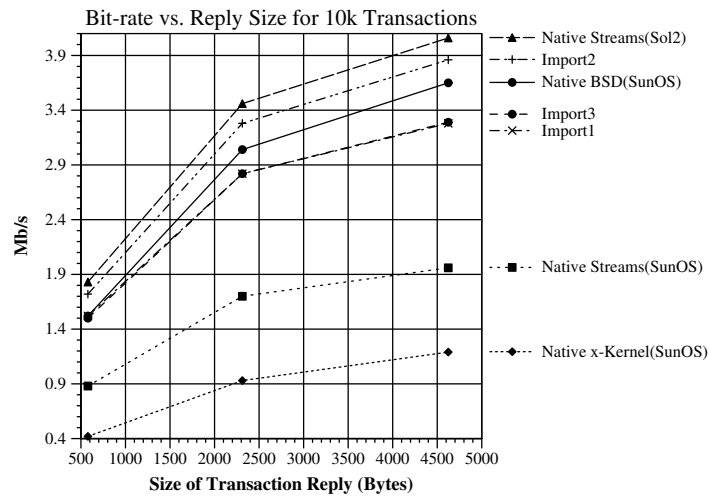


Figure 5: Protocol Graph Performance in Loopback Mode

For throughput measurements, we used a simple client and server program. The client requests a null transaction be performed by the server and measures the elapsed time. The server simply receives the transaction request and immediately sends a fixed length reply. Figure 5 compares the overall transaction throughput across the various native and imported protocol graphs we investigated. As can be expected, the native Str(Sol2) protocol graph performed best. The native Str(SunOS) and x-Kernel(SunOS) protocol graphs performed worst. The fact that the Str(Sol2) protocol graph performs better than Str(SunOS) is attributable to the considerable operating system and protocol subsystem tuning that Str(Sol2) has received. The poor performance [14] of Str(SunOS) is primarily attributable to the lack of tuning that the entire Streams subsystem underwent in SunOS 4.1.x while the poor performance of x-Kernel(SunOS) (using the x-Kernel version 3.2) is due the underlying threads package (SunOS LWP).

The performance of the hybrid protocol graphs when compared to the native Str(Sol2) and BSD(SunOS) is more interesting. The native BSD(SunOS) protocol graph performs better than two hybrid graphs (Import3 and Import1) but *worse* than one hybrid (Import2) and the native Str(Sol2). In order to explain its overall performance, we must understand the performance of the individual components that make up our protocol encapsulation approach. We next consider why our overall measurements are the way they are by examining protocol processing times and the performance of subsystem encapsulation modules.

## 5.2 Performance Discussion

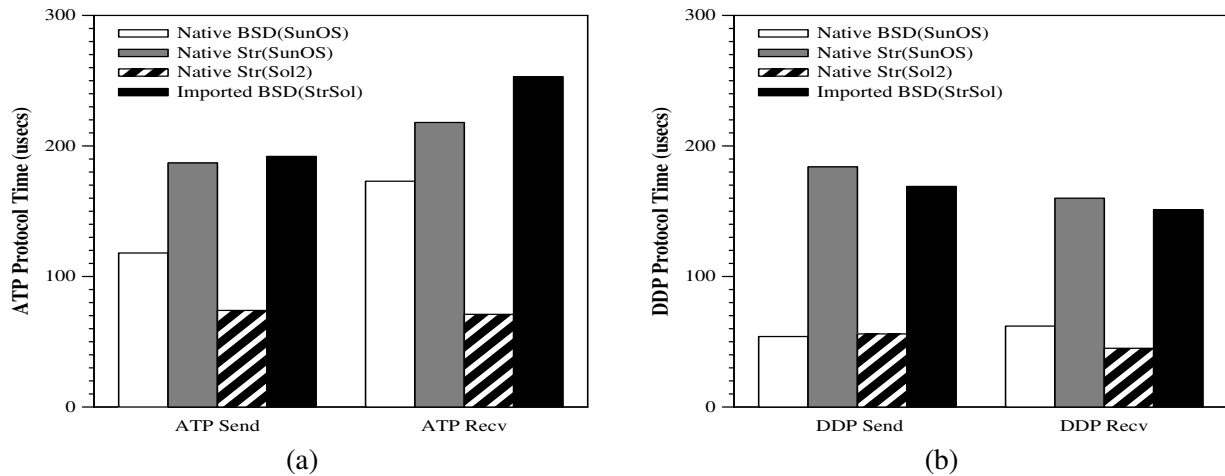


Figure 6: ATP and DDP Protocol Processing Times

Figures 6a and 6b compare the send and receive protocol processing time for the ATP and DDP protocols respectively across the various subsystems and configurations we examined. As can be seen, Str(Sol2) implementations perform as well as or better than BSD(SunOS). For simpler, datagram protocols like DDP BSD(SunOS) and Str(Sol2) are roughly equivalent while for more complex, transaction protocols like ATP, Str(Sol2) performs better. As noted previously, there is a sizable difference between the performance of Str(SunOS) and Str(Sol2) protocols; this difference is attributable to the tuning that Streams and Solaris have received. The performance difference between imported and native protocol implementations is roughly 3 : 1 in this case.

Because the Solaris2 operating system does not contain an implementation of the BSD subsystem, no direct comparison between imported and native BSD implementations can be made. Further, directly comparing BSD(SunOS) to Str(Sol2) implementations ignores those differences attributable to operating system tuning and compiler differences. However, comparing the performance of native Str(Sol2) and imported BSD(StrSol2) permits us to compare the performance of an imported protocol against its native implementation while still keeping compilers and operating systems constant. Presumably then, this difference offers us a glimpse of the performance advantage that could be gained if the target protocol were *ported* instead of *imported*. However, the software engineering community has not extensively examined software re-engineering of operating system or protocol software so it is difficult to examine the tradeoff between porting and importing. Our own experience suggests that the porting process is expensive and time-consuming.

Given the 3 : 1 difference between native and imported protocol processing time, where does all the performance go? To answer that question, we measured the performance of the individual components involved in importing a protocol: the bare protocol itself and the encapsulation module components “above” and “below” the protocol. Figure 7a depicts the components and the processing times for each. Several things should be noted. First, the overall time as measured by one series of tests is roughly 20  $\mu$ -seconds less than the sum of its parts (e.g. for ATP send processing,  $203 + 20 \approx 71 + 114 + 41$ ). This 20  $\mu$ -second difference

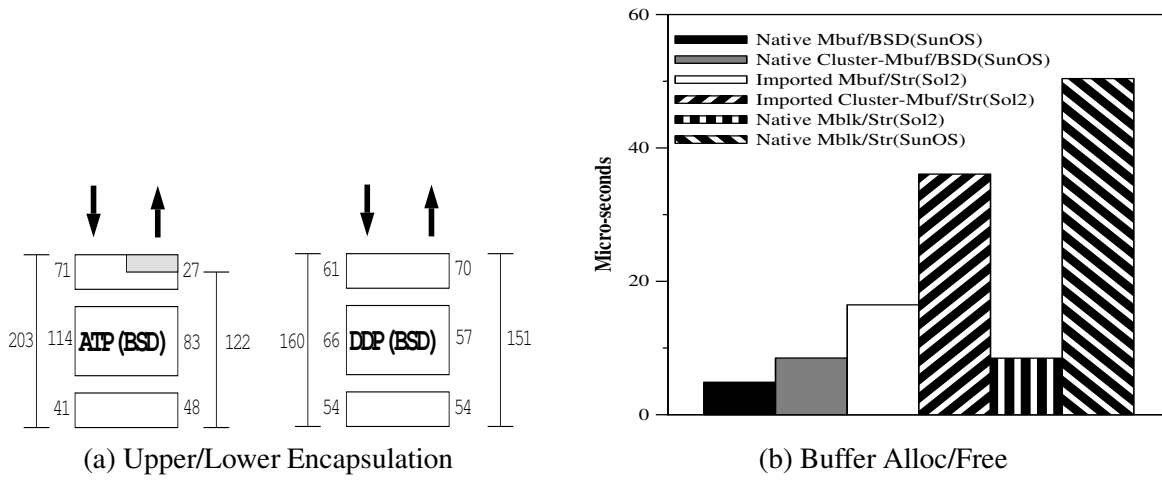


Figure 7: Encapsulation Module Component Performance ( $\mu$ -seconds)

can be attributed to the overhead added by the extra calls to the hardware's high-resolution,  $\mu$ -second timer. (We independently measured this overhead to be roughly 5  $\mu$ -seconds with favorable instruction caching.) Second, the performance of the imported protocols (once a packet traverses the protocol encapsulation layer) is roughly equivalent, in terms of protocol processing, to their native BSD(SunOS) implementations. Third, the protocol encapsulation layer components (on average) require 57  $\mu$ -seconds to traverse. This overhead is primarily due to BSD-Streams interface and *mbuf*-Streams *mblk* translations. The cost for ATP's upper encapsulation-layer is only 27 because some of the cost of that translation is assessed to the Socket-TLI crossing rather than protocol encapsulation. The cost for crossing DDP's upper encapsulation layer would be similar if the packet were destined for a user process instead of another protocol.

To understand the potential performance impact that our *mbuf* encapsulation scheme has on our imported protocols, we measured the time necessary to allocate and free 10,000 buffers in the various subsystems we have worked with. We measured the allocation/deallocation time for both small and cluster *mbufs* in SunOS and Solaris. We also measured the time to allocate/deallocate native Streams *mblks* of comparable sizes so that we may have a basis for comparison. Those results, depicted in Figure 7b, indicate that the performance impact is small. Encapsulated *mbufs* take about 12  $\mu$ -seconds and 24  $\mu$ -seconds more for small and cluster *mbufs* respectively. This overhead is only incurred at allocation and deallocation; once an *mbuf* has been allocated, access and manipulation incur no additional overhead. Also, this overhead is mostly absorbed by the upper and lower encapsulation modules rather than directly by the imported protocol itself because the protocols do not allocate and deallocate *mbufs* very often.

Given the costs of the various components, why then does Import2 perform better than the native BSD(SunOS) protocol graph? There are several reasons. First, in order to maintain consistency with previous results, throughput measurements were obtained using a client and server on the same machine (loopback mode). Therefore, the costs of the DDP's lower encapsulation module did not enter into consideration since packets destined for the same system never traversed the lower encapsulation layer. Second, the Import2 protocol

graph combines a native Str(Sol2) implementation of ATP with an imported BSD DDP implementation. That native Str(Sol2) ATP implementation so out-performs the native BSD(SunOS) ATP implementation that much of the overhead of the encapsulation is nullified (see Figure 6a). Finally, Import2 executes within the better performing Solaris 2.4 instead of SunOS 4.1.x. This performance advantage leads us to conclude that choosing the “right” combination of native and imported protocols can affect the overall performance. Comparing encapsulated and native protocol performance in isolation is not sufficient.

## 6 Comparing Protocol Encapsulation to Other Approaches

When undertaking the task of introducing additional protocols into an existing environment, several approaches can be taken. First, the protocols can be developed from scratch or, if available, their implementations for other subsystems can be ported. Second, the subsystems for which the additional protocols were originally developed can themselves be ported instead. Lastly, if both subsystems exist within the given host, then subsystem adaptation [14] could be used instead of protocol encapsulation. In this section, we compare these alternatives. Although it is impossible to provide an entirely quantitative analysis, this comparison and discussion does provide insight into the tradeoffs associated with each approach. We focus on the implementation costs in terms of complexity and required expertise in addition to the overall performance of each approach.

### 6.1 Protocol Encapsulation vs. Protocol Porting

Given that many of today’s protocol implementations are ported instead of developed originally, it is instructive to compare protocol porting and our protocol encapsulation approach. Our comparison centers around two main questions. First, how does the expense (in terms of time and expertise) of developing a protocol encapsulation module compare with the expense of porting a protocol implementation from one subsystem to another? Second, given that the cost of implementing a protocol encapsulation module is less than protocol porting, what is the performance overhead, beyond which protocol porting becomes more attractive than protocol encapsulation?

Our comparison of the overheads of protocol porting and subsystem encapsulation development assumes the availability of “ready-made” protocol encapsulation modules which are copied and then specialized to accommodate the particular protocol being imported. For example, when a protocol programmer wishes to import a BSD protocol into Streams, that protocol programmer would first copy the “generic” BSD protocol encapsulation module and then specialize it for the target protocol in question. The question then, is how difficult and expensive, relative to protocol porting, is this specialization process?

Some insight into the relative difficulty can be had by examining the number of new lines of C-code associated with the task of porting a protocol implementation relative to that needed when specializing protocol encapsulation modules. For example, during the course of our work we determined that about 700 lines of C-code were necessary to specialize our ATP/BSD protocol encapsulation module for use with our DDP protocol. In contrast, when porting ATP and DDP, we had to change about 800 to 1200 lines of C-code. Two things should be noted. First, our ATP protocol encapsulation module was used as the “base”

from which our second encapsulation module was constructed. Therefore, the difference in code represents the approximate quantity of specialization necessary to take a basic “ready-made” protocol encapsulation module and use it with another protocol. In the case of DDP’s encapsulation, the 700 lines of code were added to the “ready-made” encapsulation module primarily to accommodate data-link layer and user-protocol control encapsulation. Analysis of other protocol implementations (e.g. the native BSD Internet implementation) leads us to believe that this amount is larger than normal and represents an upper bound on the amount of specialization required for other protocols. Second, the specialization we performed was undertaken without any language or operating system support for or encouragement of re-use. For example, had our protocol development been performed using a language providing better structure for code-reuse (e.g. C++), our specialization efforts may have required fewer lines of code.

Another item worth mentioning is the amount of protocol subsystem knowledge necessary to port a protocol implementation versus that necessary to specialize an encapsulation module. In order to port a protocol implementation from subsystem to subsystem, in depth knowledge of both subsystems is required. In particular, detailed knowledge about the host subsystem and its differences from the target subsystem must be known. In contrast, “ready-made” encapsulation modules already encode a large portion of that knowledge. For protocol encapsulation, only knowledge about the target protocol and its protocol-specific interaction with the target subsystem is necessary.

Finally, one must compare the performance and implementation cost tradeoffs involved when choosing protocol encapsulation or protocol porting. Our protocol encapsulation approach navigates the fine line between performance and porting costs. While the performance trade-off is not large (approximately 110-120  $\mu$ -seconds per packet of additional protocol processing by the encapsulation module), the protocol programmer must decide whether that cost is less than the protocol porting cost. While we believe that the cost is negligible, some applications may require the highest possible performance and may be willing to pay the greater cost of protocol porting to obtain it. However, in the short-term or in transition periods (while a native implementation or protocol port is taking place), protocol encapsulation makes considerable sense.

## 6.2 Protocol Encapsulation vs. Subsystem Porting

Intuitively, the cost of porting an entire protocol subsystem is significantly greater than that of encapsulating a single protocol. But, if we view the cost of encapsulating protocols as linear with respect to the number of protocols, at some point, the cost of porting the protocol subsystem will be less than the total cost of encapsulating each individual protocol. With that threshold in mind, we compare and contrast subsystem porting and protocol encapsulation.

To some extent, subsystem porting and protocol encapsulation are different degrees of the same approach. However, a distinction can be drawn between them in terms of how each approach is integrated with the host operating system. Figure 8 depicts the relationship between the operating system, protocol subsystem, and protocol encapsulation. When a subsystem is ported, it is directly integrated into the host operating system; when a protocol is encapsulated, it is directly integrated into the host protocol subsystem and only indirectly with the host operating system. Whereas a ported subsystem can exist *without* the aid of the

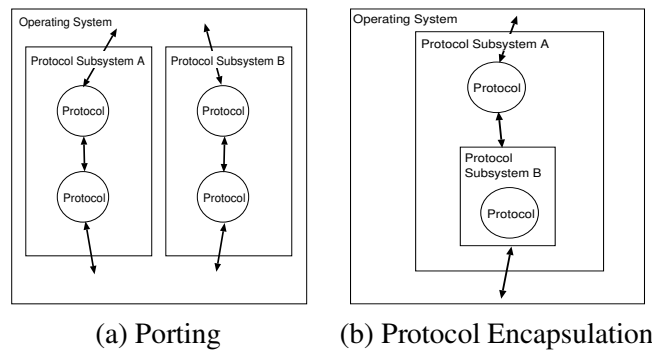


Figure 8: Subsystem Porting vs. Protocol Encapsulation

host subsystem, protocol encapsulation cannot exist without the host subsystem because it utilizes the host subsystem's services in order to provide the services of the target subsystem.

Given this distinction then, what are the tradeoffs involved when choosing between protocol encapsulation and subsystem porting? In order to port a subsystem, one must have detailed knowledge of the target subsystem as well as the host operating system. Substantial amounts of operating systems programming may be involved in addition to the actual porting of the subsystem code. For example, the interface between a protocol subsystem and application programs may require the addition of new system calls and library routines. Their development and integration with the host operating system may not be trivial. While modern operating systems support this extensibility, many existing systems do not. Further, subsystem porting reinforces the phenomenon of “ships in the night” because the level of integration between each subsystem is usually nil. Protocols in one subsystem cannot normally utilize the services of protocols in other subsystems without the introduction of additional code (e.g. subsystem adapters [14]). Another issue is that subsystem porting imposes the burden of additional testing; the newly ported subsystem must be tested along with the particular protocols in question. With protocol encapsulation, testing requirements are more constrained and are restricted to only those protocols being imported. As with protocol encapsulation vs. protocol porting, some applications' performance may be so important as to necessitate subsystem porting instead of protocol encapsulation.

Another issue is that of the scale at which protocol encapsulation should occur. That is, should one provide an encapsulation module around each protocol or should one encapsulate an entire protocol graph? The answer depends on the level of integration with the host subsystem that the protocol programmer requires. For example, if a protocol programmer wants to import a particular suite of protocols for a particular application, but never plans on making use of the individual protocols separately, it makes more sense to encapsulate the entire protocol graph rather than each individual protocol. The overall performance would almost certainly improve because the number of translations (encapsulation boundary crossings) would be reduced and what translation costs are incurred could be amortized over the entire protocol graph instead of only a single protocol. On the other hand, if the protocol programmer wishes to import a suite of protocols and make the individual members available to other protocol graphs and applications, then each protocol

should be encapsulated separately. Encapsulation on the protocol-graph scale more closely resembles subsystem porting but is distinguishable because it requires less integration with the host operating system.

### 6.3 Subsystem Adaptation vs. Protocol Encapsulation

In our previous work [14], we addressed the difficulties encountered when porting protocol implementations and developed an approach which allowed protocol implementations residing in different subsystems to be combined into a single protocol graph (termed a multi-subsystem protocol graph). That approach took advantage of situations in which multiple protocol subsystems are supported by a given host. Although our current work focuses on systems in which only a single protocol subsystem exists, it could also be applied to those containing multiple subsystems. Indeed, with our current and previous approaches, a protocol programmer facing the task of porting a protocol implementation would now have two choices: leave the protocol in its native subsystem and utilize a subsystem adapter to build a multi-subsystem protocol graph or utilize a protocol encapsulation module to import the unmodified protocol code into the protocol programmer’s subsystem of choice. These choices are depicted in Figure 9.

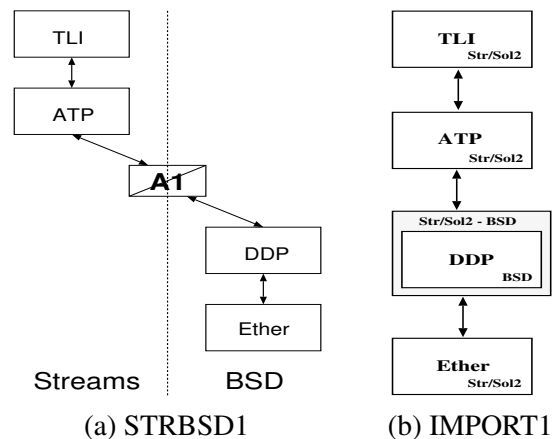


Figure 9: Subsystem Adaptation vs. Subsystem Encapsulation

The adapter in Figure 9a (see [14]) required approximately 1900 lines of C-code versus about 1200 lines of C-code for the protocol encapsulation module depicted in Figure 9b. This comparison is not exact, but does provide a rough measure of the implementation costs of each. Both approaches require some subsystem-specific knowledge of the protocol programmer but less so than that required to port a protocol implementation from one subsystem to another. Further, because both adapters and encapsulation modules pre-encode many of the subsystem differences, the protocol programmer generally need only be concerned with converting protocol-specific control operations and converting protocol-specific arguments and options. Consequently, we believe the implementation costs for each approach are equivalent.

We next consider the send and receive protocol processing times for each approach. Adaptation (A1 in Figure 9a) costs approximately 145 and 280  $\mu$ -seconds of protocol processing time for sending and receiving

while the comparable protocol encapsulation costs 115 and 124  $\mu$ -seconds respectively. Initially, it would appear that our encapsulation approach is more efficient than subsystem adaptation. However, a direct comparison of the performance costs incurred by each approach is difficult due to two factors. First, the underlying operating systems used for each of the protocol graphs (STRBSD1 and IMPORT1 in Figure 9) is different: STRBSD1 is contained within SunOS 4.1.3\_U1 while IMPORT1 exists in Solaris 2.4. Second, our previous work has demonstrated that the Streams subsystem has been extensively tuned in Solaris 2.4 when compared with SunOS 4.1.3\_U1. Consequently, we cannot conclude that either approach is better, in terms of performance, than the other. However, we *can* conclude that using our protocol encapsulation approach to move protocol code from a less-tuned operating system (BSD in SunOS 4.1.x) to a more tuned protocol subsystem and operating system (Streams in Solaris 2.x) can have its performance advantages. The overall throughput for protocol graph IMPORT2 in Figure 5 corroborates this assertion. That protocol graph, which uses importation to move BSD protocol code from SunOS 4.1.x to Streams in Solaris 2.x, obtains better overall performance than its native BSD/SunOS counterpart.

Finally, in order to compare subsystem adaptation and subsystem encapsulation, one must examine the ability of each approach to preserve the correct functionality of the original implementation. One functional mismatch between Streams and BSD involves protocol graph construction and modification. Streams provides the ability to dynamically create and modify the protocol graph at run time while BSD provides only a very limited amount of support. When encapsulating BSD within Streams, this mismatch is not important because Streams' functionality encompasses that of BSD. On the other hand, when attempting to import a Streams protocol into BSD, that potential mismatch may be important enough to prevent it. In this case, subsystem adapters may be preferable because protocols remain in their native subsystems and, therefore, can avoid the functional mismatch altogether.

Another potential functional mismatch involves each approach's ability to preserve a subsystem's process architecture. For example, BSD protocols use a vertical process architecture while Streams uses both horizontal and vertical. Because Streams supports the process architecture used by BSD (vertical), no functional mismatch occurs. On the other hand, when importing Streams protocols into BSD, the protocol encapsulation module must emulate the horizontal process architecture. This emulation could potentially lead to a functional mismatch that might prevent the correct operation of the Streams protocol. Again, subsystem adapters may be preferable because protocols can remain in their native subsystem and avoid potential functional mismatches.

Still another potential functional mismatch involves the flow of control information between protocols. For both approaches, however, control information must be translated as it flows between subsystems; this translation may introduce functional mismatches. For example, certain Streams control operations for inter-module flow-control are not supported in BSD. In this case, however, this mismatch is normally insufficient to prevent the correct functioning of the protocol graph. Nevertheless, neither approach offers an obvious advantage over the other.

## 7 Conclusion

We have introduced protocol encapsulation modules, through which protocol programmers can take protocol code written for one subsystem and use it un-modified in another. We presented the overall approach and then discussed its application to the BSD and Streams protocol subsystems. We then analyzed its performance and compared it with other approaches. Our measurements indicate that for a relatively small performance cost, protocol programmers can easily, and without modification, incorporate protocol code written for different subsystem into their current environment. Although functional mismatches between the two subsystems can exist, our approach has been successfully applied to a suite of operational protocols.

## References

- [1] M. B. Abbott and L. L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, February 1993.
- [2] Robert Allen and David Garlan. Formalizing architectural connection. In *ICSE-16. 16th International Conference on Software Engineering*, pages 71–80.
- [3] Guillermo Arango and Rubén Prieto-Díaz. Domain analysis concepts and research directions. In Guillermo Arango and Rubén Prieto-Díaz, editors, *Domain Analysis and Software Systems Modeling*, pages 9–26, 1991.
- [4] AT&T. *UNIX System V Network Programmer's Guide*. Prentice-Hall Inc., 1987.
- [5] K.L. Calvert and S.S. Lam. Adaptors for protocol conversion. In *Proceedings IEEE INFOCOM '90*, volume 2, pages 552–60, 1990.
- [6] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications*, 27(6):23–29, June 1989.
- [7] Russell J. Clark, Mostafa H. Ammar, and Kenneth L. Calvert. Multi-protocol architectures as a paradigm for achieving inter-operability. In *Proceedings of IEEE INFOCOM*, pages 136–143, 1993.
- [8] Russell J. Clark, Kenneth L. Calvert, and Mostafa H. Ammar. On the use of directory services to support multi-protocol inter-operability. In *Proceedings of IEEE INFOCOM*, pages 784–791, 1994.
- [9] S. H. Goldberg and J.A. Mouton. A base for portable communications software. *IBM Systems Journal*, 30(3):259–79, 1991.
- [10] Paul E. Jr. Green. Protocol conversion. *IEEE Transactions on Communications*, 34(3):257–268, March 1986.
- [11] IEEE. Information technology-portable operating system interface (POSIX)-part 1: System application program interface (API) [C language. Technical Report Std 1003.1-1990, IEEE/ANSI, 1990.
- [12] IEEE. Information technology-portable operating system interface (POSIX)-protocol independent interfaces. Technical Report Draft 6.1 Std 1003.1g, IEEE/ANSI, 1995.
- [13] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in TCP/IP. *ACM SIGCOMM-1993 Symposium*, pages 259–268, September 1993.
- [14] Bobby Krupczak, Mostafa Ammar, and Ken Calvert. Multi-subsystem protocol architectures: Motivation and experience with an adapter-based approach. In *Proceedings of IEEE INFOCOM*, March 1996.
- [15] S.S. Lam. Protocol conversion. *IEEE Transactions on Software Engineering*, 14(3):353–62, March 1988.

- [16] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quaterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, 1st edition, 1989.
- [17] Sun Microsystems. *STREAMS Programmer's Guide*. Sun Microsystems, Inc., August 1994.
- [18] Sun Microsystems. *Writing Device Drivers*. Sun Microsystems, Inc., August 1994.
- [19] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10:110–143, May 1992.
- [20] Christos Papadopoulos and Gurudatta Parulkar. Experimental evaluation of SunOS IPC and TCP/IP protocol implementation. *IEEE/ACM Transactions on Networking*, 1(2):199–216, April 1993.
- [21] Craig Partridge and Stephen Pink. A faster UDP. *IEEE/ACM Transactions on Networking*, 1(4):429–440, August 1993.
- [22] James M. Purtilo and Joanne M. Atlee. Module reuse by interface adaptation. *Software – Practice and Experience*, 21(6):539–556, 1991.
- [23] M.D. Rice and S.B. Seidman. A formal model for module interconnection languages. *IEEE Transactions on Software Engineering*, 20(1):88–101, January 1994.
- [24] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, October 1984.
- [25] D.C. Schmidt. A C++ wrapper for unix I/O multiplexing: the object-oriented design and implementation of the reactor. *C++ Report*, 5(7):32–43, 1993.
- [26] Douglas C. Schmidt and Tatsuya Suda. Transport system architecture services for high-performance communications systems. *IEEE Journal on Selected Areas in Communications*, 11(4):489–506, May 1993.
- [27] Gursharan S. Sidhu, Richard F. Andrews, and Alan B. Oppenheimer. *Inside AppleTalk*. Addison-Wesley, 1st edition, 1989.
- [28] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, October 1993.
- [29] X/Open. X/Open transport interface (XTI), version 2. Technical Report C318, X/Open, August 1993. Becomes C523 Networking Services, Issue 4, Version 2 12/96.