Protocol Portability through Module Encapsulation*

Bobby Krupczak  
Ken Calvert  
Mostafa Ammar  
College of Computing  
Georgia Institute of Technology  
Atlanta, GA 30332-0280  
{rdk,calvert,ammar}@cc.gatech.edu  
http://www.cc.gatech.edu/computing/Telecomm/playground/

Abstract

Because protocol software is difficult and expensive to implement and test, it is often ported between systems instead of rewritten from scratch. Unfortunately, porting protocol software can be as difficult as from-scratch development, due to inherent differences in subsystem design. Thus, protocol subsystems can have a profound effect on the portability of a protocol implementation. 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.

1. Introduction

The environment within which a protocol is developed can have a profound effect on its implementation and its portability. Indeed, protocol code written for one system is often unusable in another. Because protocol software is complex and difficult to implement and test, it is usually developed within protocol subsystems (e.g. Streams [13], BSD [9], or the x-Kernel [10]). 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

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

services they provide, and the structure they impose. This difficulty, the 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 protocol subsystems only exacerbates the problem as protocol programmers are forced to port or re-implement existing protocols.

Our previous work [7] also examined the difficulties encountered when porting protocol implementations by focusing on environments in which multiple protocol subsystems are supported. It allows protocols residing in different subsystems to be combined (without modification or porting) into a single protocol graph (termed a multi-subsystem protocol graph). 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 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

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 speci-
fication 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, with protocols represented as nodes and their interconnection by edges.

Protocols interface with the subsystem and other protocols to provide communication services to other protocols or users; they 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. We define importation as the overall process which takes protocol code from the target subsystem and embeds it within the host subsystem.

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.

Clark et al[5] 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) [6, 3]. However, protocol conversion abstracts away from a protocol's implementation and its subsystem.

Others [4, 15, 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.

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. Module interconnection languages [12, 11] 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 [2] 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 (the host subsystem). Because we wish to avoid modifying the original protocol implementation (as well as the host subsystem), we must 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 briefly present the overall functionality of protocol encapsulation modules. For a more complete treatment, please refer to [8].

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.

A protocol encapsulation module emulates a native protocol to the host subsystem, while 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 target protocol itself). Second, it augments the host's subsystem to provide those services present in the target subsystem but not the host. 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 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.\(^1\)

Among the most important determinants of a protocol encapsulation module’s functionality are the different protocol models used by the target and host subsystems as well as the support services they provide. 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. Example interfaces include those for opening and closing a session, and sending and receiving data. In addition, the protocol model defines the manner in which protocol entities like messages, layers, connections, and buffers are bound to the underlying unit of scheduling (e.g., horizontal or layer-scheduling and vertical or connection-scheduling). The protocol model also defines protocol graph connectivity options, which include support for layering, graph construction, and graph alteration.

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 unmodified 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 [14] implementation. The AppleTalk suite of protocols provide a set of services similar to the Internet family and is in widespread use; further, it is not biased towards or against the BSD and Streams subsystems. The BSD and Streams subsystems 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. 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. Finally, this environment solves a real problem, namely the need to incorporate BSD-coded protocols within the Streams subsystem.

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.

Protocol Model Differences. The BSD protocol model defines several things, each different from that defined by Streams. They include protocol methods and functions, timer processing, and protocol graph connectivity options. For the sake of brevity, we only consider timer processing and refer the reader to [8] for a more in-depth analysis. The BSD subsystem defines methods for fast and slow timeout invocation that are invoked approximately every 250 and 500 milliseconds respectively while Streams defines service queues. Both are used to schedule and perform periodic processing or "housekeeping" chores such as re-transmitting packets, clearing internal state information, or probing the status of a peer protocol. However, service queues and their accompanying service methods are scheduled non-deterministically and are, therefore, a poor match for the fast (250 millisecond) and slow (500 millisecond) timeout methods. While matching service queues to fast and slow timeout methods may be syntactically possible, it may 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.

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

\(^1\)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.
translate all data and control information as it passes to and from the Streams subsystem as well as to and from other Streams protocols. 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 and takes the optimistic approach by ignoring all others. Fortunately, these operations are either not applicable (e.g. hangup) or seldom used (e.g. start and stop).

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.

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.

mbuf Support. Although the Streams subsystem provides its own buffer support (Streams mblk), our BSD protocol encapsulation module cannot simply translate mbuf calls to their equivalent Streams mblk calls for several reasons. First, both mbufs and Streams mblkks do not support a strict interface that hides the protocol programmer from their internal implementations. Second, BSD 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. 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 mblkks. When converting from mbufs and Streams mblkks, only pointers need be adjusted. When converting from Streams mblkks to mbufs, only an additional (hopefully cached) and empty mbuf 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. We measure the performance impact of our mbuf encapsulation scheme in a later section.

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. 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. Therefore, the encapsulated socket layer need only support the socket structure and a few function calls to manipulate it.

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 1). During the course of our work, we developed two protocol encapsulation modules and 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, and the encapsulation modules and their sub-components. As a baseline, we compare (where appropriate) our results from those obtained in earlier work [7].

![Protocol Graphs](image)

**Figure 1. Native and Hybrid Protocol Graphs**

All our measurements were taken on a Sun SPARCstation-LX running in single-user mode. For Solaris measurements, denoted using Sol2, we used Solaris 2.4; for SunOS measurements, denoted SunOS, we used SunOS 4.1.3.U1. 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. We refer to protocol graphs containing some native and some imported protocols implementations as "hybrids".

![Bit-rate vs. Reply Size for 10k Transactions](image)

**Figure 2. Protocol Graph Performance in Loopback Mode**

For throughput measurements, we used a simple client and server that exchange a simple transaction and measure the time difference. Figure 2 compares the overall transaction throughput across the various native and imported protocol graphs we investigated. The native Str(Sol2) protocol graph performed best while the native Str(SunOS) and x-Kernel(SunOS) 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 [7] 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 primarily to 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).

![Protocol Processing Times](image)

**Figure 3. ATP and DDP Protocol Processing Times**

In order to explain the overall performance, we must understand the performance of the individual components that make up our protocol encapsulation approach. Figures 3a and 3b 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). The performance difference between imported and native protocol implementations is roughly
3:1 in this case. Because Solaris2 does not contain an implementation of the BSD subsystem, no direct comparison between imported and native BSD implementations can be made. However, comparing the performance of native Str(Sol2) and imported BSD(StripSol2) 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.

![Graph showing performance comparison]

(a) Upper/Lower Encapsulation Components

![Graph showing buffer allocation and free]

(b) Buffer Alloc/Free

Figure 4. Encapsulation Module Component Performance

Given the 3:1 difference between native and imported protocol processing time, where does 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 4a depicts the components and the processing times for each. 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 while each protocol encapsulation layer component (on average) requires about 57 μ-seconds to traverse. This overhead is primarily due to BSD-Streams interface and mbuf-Streams mblk translations.

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 (mbuf and Streams mblk) buffers in the various subsystems we have worked with. Those results, depicted in Figure 4b, indicate that the performance impact is small. Encapsulated mbufs take about 12 μ-seconds and 24 μ-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. Given the costs of the various components, why then does Import2 perform better than the native BSD(SunOS) protocol graph? 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 3a). Import2 also 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. Third, if both subsystems exist within the given host, then subsystem adaptation [7] 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 provides insight into the tradeoffs associated with each approach.

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 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. Our comparison centers around two main questions: 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 and what is the performance overhead, beyond which protocol porting becomes more attractive than protocol encapsulation?

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 "ready-made" protocol encapsulation module for use with our DDP protocol. 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. In contrast, when porting ATP and DDP, we had to change about 800 to 1200 lines of C-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 μ-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.

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.

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. 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 [7]). Subsystem porting also 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.

The scale (ranging from an individual protocol to an entire protocol graph) at which protocol encapsulation should occur is also important. 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.

Protocol Encapsulation vs. Subsystem Adaptation. In our previous work [7], 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). For the purposes of our comparison (with our current importation-based approach), we consider two functionally equivalent protocol graphs, one utilizing an appropriate adapter module (STRBSD1 in [8], and the other (IMPI in Figure 1) utilizing an appropriate protocol encapsulation module. For the sake of brevity, we limit our comparison to performance and correctness and omit code complexity since the two approaches appear roughly equivalent in that category.

Because our two functionally equivalent protocol graphs are
not contained within the same operating system (one is in
SunOS 4.1.3.U1 and the other in Solaris 2.4), we cannot
conclude that either approach is better, in terms of perfor-
man ce, 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 advan-
tages. The overall throughput for protocol graph IMPORT2
in Figure 2 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.

For our comparison, we must also examine the ability of each
approach to preserve the correct functionality of the original
implementation despite functional mismatches that may oc-
cur between protocol subsystems. Potential mismatches can
occur between each subsystem’s protocol graph construc-
tion, between each subsystem’s process architecture, and
in how control information propagates between protocols
in their respective subsystems. For differences in proto-
col graph construction and process architecture differences,
subsystem adapters may be preferable because protocols re-
main in their native subsystems and, therefore, can avoid any
mismatch altogether. Lastly, for control flow, mismatches
are normally not sufficient to prevent the correct functioning
of the protocol graph; 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 ap-
lication 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
different subsystem into their current environment. Al-
though functional mismatches between the two subsystems
can exist, our approach has been successfully applied to
a suite of operational protocols.

References

to protocol implementation. IEEE/ACM Transactions on Net-

concepts and research directions. In Guillermo Arango and
Rubén Prieto-Díaz, editors, Domain Analysis and Software

In Proceedings IEEE INFOCOM ’90, volume 2, pages 552–
60, 1990.

[4] David D. Clark, Van Jacobson, John Romkey, and Howard
Salwen. An analysis of TCP processing overhead. IEEE

Multi-protocol architectures as a paradigm for achieving inter-
143, 1993.


[7] Bobby Krupczak, Mostafa Ammar, and Ken Calvert. Multi-
subsystem protocol architectures: Motivation and experience
INFOCOM, March 1996.

portability through module encapsulation. Technical Report
GIT-CC-96-12, College of Computing, Georgia Institute of
Technology, April 1996.

[9] Samuel J. Leffler, Marshall Kirk McKusick, Michael J.
Karels, and John S. Quarterman. The Design and Implemen-
tation of the 4.3 BSD UNIX Operating System. Addison-Wesley,

[10] S. W. O’Malley and L. L. Peterson. A dynamic network ar-
chitecture. ACM Transactions on Computer Systems, 10:110–

interface adaptation. Software – Practice and Experience,

interconnection languages. IEEE Transactions on Software

[13] D. M. Ritchie. A stream input-output system. AT&T Bell Lab-

[14] Gursharan S. Sidhu, Richard F. Andrews, and Alan B. Oppen-
heimer. Inside AppleTalk. Addison-Wesley, 1st edition,
1989.

Implementing network protocols at user level. IEEE/ACM