Multi-Subsystem Protocol Architectures: Motivation and Experience with an Adapter-Based Approach *†

Bobby Krupczak
Mostafa Ammar
Ken Calvert
{rdk,ammar,calvert}@cc.gatech.edu

GIT–CC–95-08

Februrary 2, 1995
Revised July 18, 1995

Abstract

Protocol software is often difficult, cumbersome, and expensive to implement and test in today's computing environments. To reduce this difficulty, several things are done: communications software is commonly subdivided into layers and organized into a protocol graph; it is developed within a protocol or networking subsystem; and it is often ported rather than developed from scratch. Inherent differences in the multitude of protocol subsystems offer a dizzying array of features, functionality, and drawbacks; their differences often reduce the portability and efficiency of protocol code. In this paper, we consider the differences in subsystems and their effect on the portability and performance of protocol implementations. We propose an approach for combining the "better" features of protocol subsystems by constructing protocol graphs composed of protocols residing in different subsystems.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

†This research is supported by a grant from the National Science Foundation (NCR-9305115) and the TRANSOPEN project of the Army Research Lab (formerly AIRMICS) under contract number DAKF11-91-D-0004.

1 Introduction

Protocol software can be difficult, cumbersome, and expensive to develop and test. In practice, several techniques are used to reduce this complexity. First, protocol code is subdivided into layers or modules and organized into a graph structure representing how those modules are combined to provide communication services. Second, protocols are developed within programming environments or *subsystems* (such as BSD [14], Streams[2, 22], or the *x*-Kernel[18, 11]) which provide an overall protocol *model* and offer organized, consistent access to operating system resources. Third, protocol code is often ported from one subsystem to another rather than developed from scratch.

A protocol's implementation is generally tightly coupled with the subsystem in which it is developed. Each subsystem places fundamental constraints on the structure of a protocol implementation and how it interacts with the subsystem; each subsystem offers its own unique advantages and disadvantages. This has two effects:

1. Protocol implementations today typically reside entirely within a single subsystem. The protocol users and developers are thus constrained by the features and drawbacks of that subsystem.
2. Introducing a new protocol implemented in some subsystem into another subsystem involves a tedious "porting" process that is made more difficult by protocol subsystem differences.

Current trends indicate that no one protocol subsystem or operating system will prevail. In fact, these problems are becoming exacerbated as new operating systems (e.g., [15]) and accompanying protocol subsystems are deployed.

In this paper we address the two issues above by allowing protocol implementations to span multiple subsystems; we call the resulting implementations *multi-subsystem architectures*. This allows the protocol programmer to take advantage of the "better" features (e.g., configurability and performance) of each protocol subsystem. In addition, the possibility of multiple subsystem protocol implementations eliminates the need to port protocols from subsystem to subsystem, thus allowing for easier introduction of new protocols.

Examples of multi-subsystem protocol implementations are shown in Figure 1. Figure 1a shows a transition architecture incorporating IPv6[4] written in Streams but not yet ported to BSD. We expect this type of architecture to allow faster adoption of new protocols. Figure 1b shows an environment supporting high performance coupled with dynamic protocol configuration. Since BSD purports better performance[1] than Streams, portions of the protocol graph not requiring configuration are kept within the BSD subsystem. To support dynamic configuration (i.e., selection) of compression algorithms 1 and 2, those parts of the graph reside in Streams. Instead of sacrificing overall performance by implementing all protocols within Streams, or sacrificing dynamic configurability by implementing all protocols in BSD, the protocol programmer can construct a multi-subsystem protocol graph combining both features.

The contributions of this paper are twofold. First, we further study the factors described above motivating our multi-subsystem architecture. We do this through a case study involving the implementation,

[1] We examine this issue later in the paper.

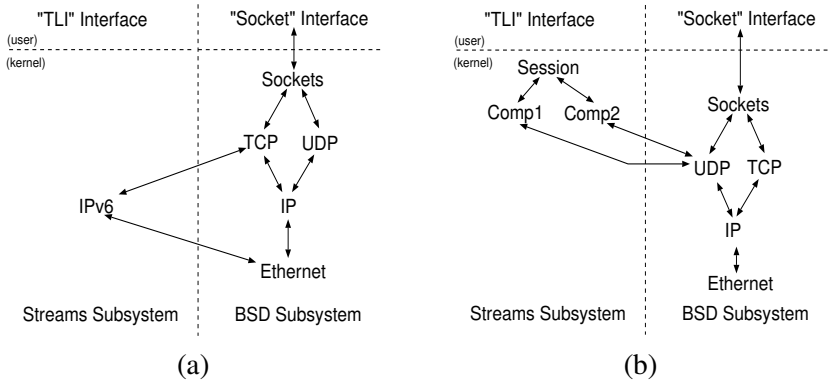

Figure 1: Multi-Subsystem Protocol Graph Examples

porting and analysis of a protocol suite in the context of three protocol subsystems: BSD, Streams, and the *x*-Kernel. Second, we propose a general approach that can be used to construct multi-subsystem protocols. We describe our experiences implementing and testing multi-subsystem protocol graphs using our approach.

The rest of the paper is organized as follows. The next section contains some background material including the definition of some terminology and a discussion of related work. Section 3 introduces our case study. Section 4 discusses the three protocol subsystems we considered from the viewpoint of protocol portability. Section 5 considers the performance aspects of protocol subsystems in the context of our case study. Section 6 introduces our adapter-based approach to constructing multi-subsystem protocol graphs and describes our experience in the design and implementation of one such adapter. The paper is concluded in Section 7

2 Background

2.1 Terminology and Definitions

For the purposes of this paper, a *protocol* is a software module that corresponds to an implementation of a traditional, monolithic protocol like TCP. Alternatively, it may correspond to an implementation of a single, atomic protocol function (e.g., header checksumming) intended for use in conjunction with other protocols. This latter type, referred to as a micro-protocol, is outlined in [18]. Protocols typically execute within the context of a *protocol subsystem*, which organizes operating system resources like buffers and timers in a manner which eases the burden of protocol development. Protocols are arranged using a graph structure (commonly referred to as a *protocol graph*) representing how they are combined to provide communication services. Figure 1 provides examples of protocol graphs. Nodes in this graph represent protocol implementations and edges represent their interconnection; they are oriented such

that protocols are connected to from above and below. Further, users are generally situated at the top and access the protocol graph through *application-interface protocols*; the actual, physical subnetwork is generally situated at the graph's bottom.

Protocols interact or 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 other protocols. *Portability*, finally, is a qualitative measure of how difficult it is to take protocol source code developed for one subsystem and place or compile it within another subsystem.

2.2 Related Work

Current protocol research focuses primarily on performance in the context of a single subsystem by either proposing new, alternative subsystems or improving the performance of existing protocol implementations. These approaches limit the choices a protocol programmer has by requiring that the protocol graph be contained entirely within one subsystem. Further, approaches that introduce new subsystems only add to the protocol portability problem. We survey current research and classify it on how it relates to portability and the interplay between the subsystem and protocol performance.

In the *Base* subsystem [9], the authors directly address the protocol portability problem. Recognizing the expense and difficulty of developing OSI protocols for several different IBM platforms (and accompanying protocol subsystems), the authors sought to eliminate the protocol portability problem by shifting the focus to the subsystem instead. They created a portable subsystem supporting protocol development. Once they ported their subsystem to a target platform, their protocol implementations could run virtually unmodified. Unfortunately, protocol programmers are restricted to *Base* and must accept its (possible) restrictions for the sake of portability.

Bodhi *et al* [16] face many of the same problems that obstruct protocol portability by addressing the problems encountered when trying to port a user-level threads package from operating system to operating system. They propose that all operating systems support a specific set of thread-related primitives that would greatly reduce the job of porting user-level threads packages. Their work is analogous to ours in that the task of porting a user-level threads package is similar to the that of porting protocol code: both are highly dependent on the types, flavors, and interfaces of the underlying services which they depend.

Thekkath *et al* [26] partially address one factor that makes protocol development difficult, namely, location of protocols inside the kernel. They argue that protocols and accompanying protocol subsystems should be developed as user-space modules. Their research focused on moving the BSD subsystem out of the kernel and into user-space. In doing so, they have taken the approach of porting the subsystem rather than porting the protocol code.

The *x*-Kernel work [18, 11] recognized the difficult nature of implementing protocols and addresses the problem by developing a subsystem with an explicit goal – the reduction of complexity of protocol programming without hampering performance. Streams [22, 2] sought to replace the traditional UNIX

method for connecting processes to terminals or network devices with a flexible, consistent subsystem supporting *modular* protocols. These approaches improve performance and reduce implementation complexity at the expense of portability and flexibility.

Still another approach to protocol implementation is the use of a special purpose protocol programming language and compiler. Morpheus ([1]) and YANC ([21]) apply complimentary approaches. Morpheus provides a general protocol compiler for implementing traditional network and transport protocols while YANC focuses on aiding the development of portable network applications. Underneath each language (and closely tied to it) resides an accompanying subsystem. As with many of the approaches already mentioned, they focus on the subsystem to address the protocol code portability problem. Once the *subsystem* has been ported, protocol code portability comes for "free". Performance gains are obtained through subsystem and compiler efficiency.

A substantial body of research focuses on interoperability through protocol conversion [10, 13]. Using this approach, interoperability is achieved by converting the protocols spoken by one or both of the communicating entities through gateways and intermediaries. Auerbach describes a protocol converter toolkit [3] while Calvert introduces protocol adaptors [5]. Although it is a related approach, protocol conversion focuses on a protocol's output (headers, messages, etc.) rather than on the environments in which protocols may operate.

Measuring and improving protocol implementation efficiency are the approaches taken in another area of protocol research. Papadopoulos and Parulkar [19] and Kay and Pasquale [12] focus on measuring operational TCP/IP implementations in order to gain a better understanding of protocol performance issues. Partridge and Pink [20] examine and improve the performance of UDP/IP within the context of the BSD subsystem. They focus on implementation efficiency rather than subsystem issues. Clark *et al* [6] examine implementation efficiency in isolation of the subsystem rather than in conjuncture. In each case, comparisons to other subsystems were not explored.

Finally, the work in [7, 8] addresses protocol interoperability concerns by proposing multi-protocol architectures in which two entities communicate by mixing and matching supported protocols until they have a common subset. Indeed, their work provides some of the motivation for our work in addressing protocol portability problems and multi-subsystem protocol architectures, as they propose to achieve interoperability by porting protocol implementations to as many machines and subsystems as possible. Reducing the complexity and difficulty of porting protocol implementations reduces the complexity and difficulty of achieving interoperability.

3 The Case Study

The insight and analysis presented in this paper is derived, in part, from a case study in which we first implemented a set of AppleTalk protocols in Streams and then ported them to the *x*-Kernel and BSD subsystems [2]. The AppleTalk protocol family was chosen (over, say, TCP/IP) since it is not biased towards or against any of the subsystems that our case study covers[3]. The AppleTalk protocols also provide a broad set of real user services similar to the Internet family of protocols and are in widespread

[2] Please refer to [25] for more information on AppleTalk.

[3] Indeed, if the AppleTalk protocol architecture is biased towards any networking subsystem, it is biased towards that contained within the Macintosh operating system.

use. We are, therefore, confident that it provides us a reasonable "platform" for considering subsystems and their effects on portability and performance. We chose the Streams, *x*-Kernel, and BSD subsystems because they represent a good combination of commercial and research subsystems. They offer a variety of features and development environments common to many subsystems.

We implemented several AppleTalk protocols during the course of our case study. The Datagram Delivery Protocol (DDP) provides traditional network layer datagram services. It is the foundation of the AppleTalk protocol family. The AppleTalk Transaction Protocol (ATP) provides sequenced, reliable delivery of transaction requests and replies and serves as the basis for distributed file and print services. We also implemented several supporting protocols: AppleTalk Echo Protocol (AEP), Routing Table Maintenance Protocol (RTMP), and the Name Binding Protocol (NBP).

Our case study proceeded in three phases. In the first phase we implemented the AppleTalk protocols in Streams and ported this implementation to both the BSD subsystem and to the *x*-Kernel[4]. Next we considered the performance of the AppleTalk protocols in the context of the various subsystems. Finally, we implemented our adapter design of the multi-subsystem architecture by combining our protocol implementations from different subsystems through the use of adapters.

It should be emphasized that the case study was used to guide our thinking and to validate our proposals. Our aim is to provide a general understanding of the issues; we believe that this case study provided an appropriate framework for such an understanding.

4 Protocol Subsystems and Portability Concerns

Subsystems shape the design and implementation of protocols by defining and imposing an *overall protocol model*, and a set of *support services* on the protocol programmer. In this section we discuss each of these two aspects with particular emphasis on their effects on the portability of protocol implementations.

4.1 Protocol Models

Each subsystem implicitly or explicitly defines a protocol model which greatly influences the way in which a protocol is implemented. The protocol model defines a *protocol's interfaces*, a *process architecture*, and defines protocol *graph connectivity options*. Because there is such a wide disparity in how various subsystems approach these aspects of the protocol model, their differences can cause considerable portability barriers. We consider each of these three aspects next.

4.1.1 Protocol Interfaces

The protocol model dictates how a protocol interfaces both to the subsystem and to other protocols. It defines both the syntactic and semantic aspects of each interface and the information that passes across

[4]Our version of the *x*-Kernel ran in user-space in the SunOS environment.

it. Typical protocol interfaces include those for opening and closing a session and sending and receiving data. Porting between subsystems first involves matching protocol interfaces in one subsystem to their corresponding interfaces in another, and then converting between their syntax and semantics. We found the following aspects of the protocol interface structure to be important:

Layer Modeling Portability problems are compounded when a subsystem imposes several distinct models. For example, BSD defines a three "layer" protocol model, drawing distinctions between network-interface protocols (e.g., Ethernet), application-interface protocols (e.g., Sockets), and everything else. When porting a protocol from Streams to BSD, one must first decide if the protocol to be ported is a network interface protocol, a user-subsystem interface protocol, or something else. In contrast, Streams and *x*-Kernel define one, single protocol model which dictates the design and implementation of all protocols implemented within each respective subsystem.

Control Interactions Protocol models may draw a distinction between control information (e.g., options and addressing information) and data. Subsystems often define specific control interactions independent of the particular protocol; protocols are generally free to augment protocol-specific control operations with those specific to a particular protocol. For example, BSD defines over twenty separate control interactions (e.g., status and address inquiries) while Streams defines roughly ten (e.g., inter-module flow control and protocol graph operations); *x*-Kernel defines none[5]. When porting protocols, the protocol programmer must match control operations where appropriate and implement those required by the new subsystem. For example, when porting from Streams to BSD, one may be required to add functionality necessary to handle those control operations defined by BSD but not Streams.

Passing Information How data and control information are passed between protocols and the subsystem is another source of portability problems. For example, BSD defines three separate protocol control interfaces; which interface is used depends on the control information's origin. The *x*-Kernel defines a single control interface used for all control interactions. In Streams control information is passed in-band through normal data interfaces; information originating from "users" (or upper protocols) is distinguished from that coming from lower protocols. During the porting process, the protocol programmer must make decisions about where data and control information enter a protocol in its native subsystem and compare and convert to those conventions defined by the subsystem that he or she is porting to.

Timing and Event Services In addition to more traditional protocol interfaces (e.g., open, close, send, and receive) protocol models can also define interfaces used for timing and event services. The manner in which each subsystem supports such functionality (e.g., connection timeouts) directly affects protocol portability and indirectly affects performance. The BSD subsystem defines two separate protocol timeout interfaces, each called at periodic intervals. Streams defines service queues and accompanying interfaces which can be scheduled for near-periodic execution. *x*-Kernel supports a general threaded

[5]The *x*-Kernel does define a control interface but places no minimal requirements.

approach whereby a protocol programmer can fork a thread to execute a well-defined function at some future time. During our case study, we found that porting a protocol written to one of these styles to another is problematic. Each style has its own performance implications as well. Non-determinism in Streams service queues can affect protocol throughput (e.g., those using sliding windows and timeouts) because a given protocol may not be able to send data as quickly as that allowed by the protocol specification. BSD timeout resolution (200 and 500 ms), on the other hand, may be too fine or too coarse for a particular protocol causing similar problems.

Implicit Interfaces and Conventions Protocol models often define implicit interfaces through the attachment of conventions to the manner in which control and data information are processed. These implicit interfaces or conventions are not generally well-specified and are only discernible by examining sample code or supporting documentation. For example, protocol models often define conventions for the allocation and disposal of buffers as well as those for byte alignment. Buffer conventions generally specify whether the caller or callee should free previously allocated buffers. Streams mandates that the last or bottom-most protocol free buffers; BSD specifies a mix between both parties; *x*-Kernel explicitly specifies that the caller free all resources.

4.1.2 Process Architecture

The manner in which protocol entities, like messages, layers, connections, buffers, and timers, are bound to the underlying unit of scheduling is often referred to as a process architecture. Many different styles exist including *vertical* and *horizontal*; within these styles can exist heavy-weight processes (in the traditional UNIX sense) or light-weight processes and threads (See [23] for an overview of process architectures and subsystems. For the sake of compatibility, we utilize their taxonomy.). In vertical process architectures, a thread or process escorts a message through the protocol graph within a single address space. Protocols invoke one another through synchronous function calls. In horizontal process architectures, protocols encapsulate one or more threads or processes: each looping through a continuous cycle of message reception, protocol processing, and message generation. Message generation and reception is generally asynchronous in nature with little or no coordination between protocols.

The BSD subsystem utilizes a single-threaded vertical process architecture in which protocol processing is performed in the context of the initiating user or at software interrupt. The *x*-Kernel utilizes a thread-per-message vertical process architecture where each message is escorted through the protocol graph by its own thread. Lastly, Streams utilizes a combination of both horizontal and vertical process architectures: Streams supports a vertical process architecture via its *put* functions and a horizontal process architecture via its service queues.

Process architecture differences can impose serious portability problems; for example, converting protocols coded to a vertical process architecture (where protocols invoke one another via synchronous function calls) to a horizontal, message-oriented process architecture may require extensive code modifications. Alternatively, converting from a vertical thread-per-message process architecture (e.g.,

x-Kernel) to that in BSD or Streams poses problems as well. For example, consider the act of converting between network layer and physical layer addresses (commonly called ARP'ing). In BSD and Streams, the general approach is to first initiate an ARP request, then save protocol state information, and return. At some future point in time (upon the reception of an ARP reply or at the expiration of a timer), the packet is sent. In a thread-per-message process architecture, no state information need explicitly be saved; after initiating an ARP request, a thread simply need suspend itself for some period of time after which it can continue processing. The code to save state and continue processing at a later time is generally very subsystem-specific and thus non-portable.

Process architecture differences can also impose serious performance problems. For example, horizontal process architectures can suffer performance penalties due to excessive context switching due mainly to their asynchronous nature [17]. On the other hand, they may perform better in environments that can effectively reduce this overhead [24].

4.1.3 Protocol Graph

Layering is such a fundamental concept in communications architectures that it is pervasive throughout their specification and implementation. First, protocol families like the Internet, OSI, and AppleTalk subdivide the task of communications into layers and corresponding protocols. Second, protocol and protocol family implementations have generally conformed to this principle by subdividing and layering their corresponding protocol code into protocol graphs. Third, subsystems have generally enforced the notion of layering by providing overall support for protocol graph construction. Variations in how the protocol graph is constructed and accessed may also pose portability and performance problems.

Protocol graphs may be constructed (and possibly destroyed) at several points in time: at protocol development time, at compile time, and at run-time. Protocol graphs may also be accessed differently. For example, a protocol graph may simply be hard coded at development-time whereby protocols directly invoke one another via function reference; alternatively, the protocol graph may be constructed at compile or run-time with protocols invoking one another via function pointer deferencing. How each protocol obtains function pointers, in the latter case, may also differ.

The protocol graph is fixed at development time in the BSD. Protocols are generally coded with explicit assumptions about their position in the protocol graph and reference their neighbors through explicit function calls. Consequently, the BSD subsystem supports little change to the protocol graph. The Streams subsystem supports the dynamic creation, modification, and destruction of protocol graphs through the exchange of control messages and function pointer dereferencing. Protocols can re-situate themselves on top of or below other protocols at any time. The mechanism for building[6] protocol graphs combines both subsystem directives (in the form of control messages) and protocol code support. The *x*-Kernel supports the creation of the protocol graph at compile-time; no mechanisms exist for its modification or manipulation without first stopping all communication and re-compiling and re-instantiating the *x*-Kernel. Protocol graphs are also built using a combination of subsystem directive and protocol code support in the *x*-Kernel.

[6] In this case, building a protocol graph includes its modification and destruction.

Porting protocol code between subsystems of each type (those supporting differing notions of the protocol graph) can also be difficult and time-consuming. Porting BSD protocols (which may explicitly assume their position in the protocol graph) to other subsystems (where these assumptions are not appropriate) involves identification and conversion of each explicit reference to neighboring protocols. The porting difficulty is compounded if a protocol makes explicit assumptions about the services provided by its neighbors. For example, because TCP_{BSD}[7] assumes that it always operates over IP_{BSD}, it passes a partially filled IP header (containing processing options) when it invokes IP_{BSD}. Therefore, porting TCP_{BSD} to a subsystem supporting dynamic protocol graph construction and in which the protocol programmer may place TCP over some non-IP protocol will involve a fair amount of modification.

Porting from a subsystem that supports some flexibility in protocol graph construction to one which does not (e.g., *x*-Kernel or Streams to BSD) also involves a fair amount of conversion or modification and may result in a reduction of services offered by a particular protocol. For example, if a newly defined protocol family specifies that its protocol graph be modifiable at run-time, porting a conforming implementation to the BSD subsystem would be difficult. The protocol programmer would be faced with the difficult task of providing dynamic protocol graph configurability in BSD without subsystem support or hard-coding each possible configuration. Alternatively, one could provide reduced communication services by supporting a subset of all possible configurations.

4.2 Support Services

In order to reduce the complexity of protocol development, subsystems often offer generic support services in the form of buffers and timers. These services are usually provided in a protocol independent manner meaning that they are not specific to any one protocol or protocol family. We discuss some of the more common services and their effects on protocol portability below.

Buffers Because communication protocols require memory buffers in varying sizes as well as the ability to arbitrarily combine them into "messages" without unnecessary data copying, most subsystems provide a special purpose memory management facility. How each subsystem approaches the implementation of this facility is another source of portability barriers.

All three subsystems provide roughly equivalent functionality such that porting protocol implementations between them offers no serious impediments. They do differ significantly, however, in their syntax. Unfortunately, since most protocols add, remove, and inspect headers stored in buffers, these differences require tedious and time-consuming conversion. Further, some subsystems (namely the *x*-Kernel) isolate the protocol programmer from the actual implementation details while others (BSD and Streams) require the protocol programmer have extensive knowledge of its implementation. For example, the *x*-Kernel provides such a convenient library of functions for buffer manipulation that the protocol programmer need not be aware of its actual implementation structure. Consequently, porting BSD and Streams protocols to the *x*-Kernel is somewhat simpler in this aspect.

[7] Our notation denotes the subsystem a protocol is developed in as a subscript; for example, TCP_{BSD} denotes the protocol TCP implemented within the BSD subsystem.

Messages, containing user data and protocol headers, are generally passed between protocols and the subsystem as arguments to protocol interfaces. Subsystems often impose conventions regarding the "shape" of messages. For example, in Streams, the first buffer in a message usually contains control and protocol options; in BSD, the first buffer in a message may contain both control and options as well as a user data and protocol headers; in the *x*-Kernel, protocol programmers convey control and options information using message "attributes". Porting between subsystems with differing conventions as to the shape of messages first involves their identification followed by their conversion. These conventions are not generally well-specified[8] and are usually poorly documented.

Other Support Services While a general buffer facility is the most common support service that subsystems offer, many do provide others. For example, BSD provides protocol independent routing table, packet queue, and timer support; the *x*-Kernel provides efficient multiplexing[9] and event (or timer) support. Streams, on the other hand, only provides support for generic packet queueing. Differences in subsystem support services introduce both syntactic and semantic porting problems. If two subsystems provide support for efficient multiplexing, their conversion is primarily a syntactic one. If one is porting a protocol from a subsystem providing such service to one that does not, one is forced to develop new code to support that lost functionality. For example, when porting protocols utilizing the *x*-Kernel's multiplexing services to Streams, one must add the necessary multiplexing code. On the other hand, when porting from Streams to the *x*-Kernel, the protocol programmer need not modify existing multiplexing code to use that provided by the *x*-Kernel.

5 Protocol Subsystems and Performance Concerns

In this section we present the results from the second phase of our case study, in which we explore the interplay between subsystems and protocol performance. We do so by measuring the time necessary to send and receive packets through separate AppleTalk protocol graphs implemented within Streams, BSD, and the *x*-Kernel. The three measurement environments[10] are shown in Figure 2. We took all measurements on a Sun SPARCstation-LX running SunOS 4.1.3C.

We took all our protocol and protocol graph measurements in loopback mode whereby a client and server residing on the same machine exchanged packets[11]. Data packets traversed the entire protocol graph with the exception of the network interface protocol (ethernet in BSD, ethernet and NIT in Streams, and ethernet, NIT, and our adapter protocol in the *x*-Kernel). All measurements were taken while the machine was in single-user mode with almost no system processes running. We took the average over multiple samples with a 95% confidence interval of at most 10% of measured values. Because our protocols have not been extensively tuned, these results measure the *relative* impact that a subsystem has on an implementation, more than raw performance.

First we present measurements of the send and receive processing time for the DDP and ATP AppleTalk protocols, and the Transport Layer Interface (TLI) and Sockets application-interface protocols.

[8] Indeed, one can hardly convey a convention using function prototypes in which the argument is simply a data pointer to a message block.

[9] Multiplexing and de-multiplexing involve the translation between protocol header information (e.g., connection or port number) and internal protocol state information associated with a user or higher layer protocol.

[10] The NIT protocol module (in the STRATALK and XKATALK protocol graphs) is necessary in order to gain access to the Ethernet device. Further, in the XKATALK protocol graph, we must also make use of our adapter protocol concept in order for protocols coded within the *x*-Kernel to send and receive ethernet packets. We elaborate in more detail on their use in a subsequent section.

[11] Although our protocols were measured in loopback mode, they are fully functional and have been tested over a network against each other, native Macintosh implementations, and routers. The source code is available via the WWW at http://www.cc.gatech.edu/computing/Telecomm/playground/playground.html

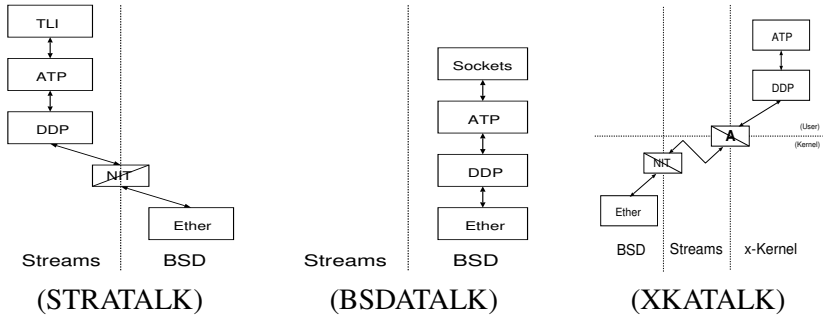

Figure 2: AppleTalk Protocol Graphs

We obtained these measurements by placing appropriate code at all entry and exit points for each protocols. By recording the entry and exit time, and their difference, we can measure the protocol processing time necessary to send or receive a packet. We average these values over 20 to 30 thousand packets. The code used to measure the protocol entry and exit times is sufficiently simple that it presents almost no impact on protocol execution time. Further, we use the hardware's high resolution clock, which provides at least microsecond accuracy. Figure 3 lists these measurements for BSD and Streams.

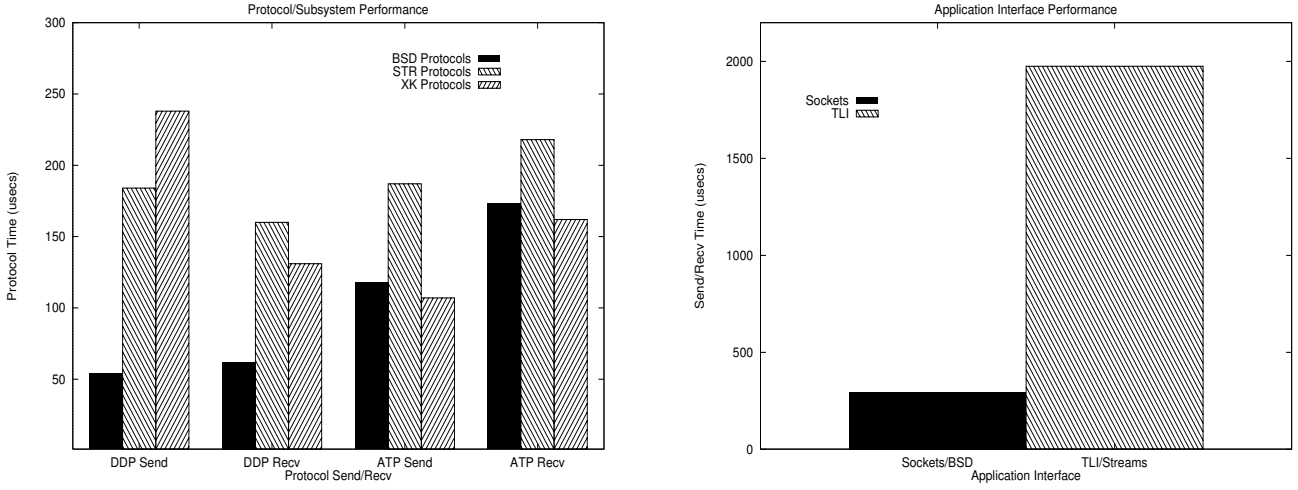

Figure 3: Protocol Timings (usecs)

Measurements for the two application-interface protocols, TLI and Sockets, were performed differently. We wrote a simple client program that sent $10,000$ 30-byte packets to itself using a protocol graph consisting of only the DDP protocol. By measuring the total elapsed time measured by the client and subtracting the total protocol execution time (confirmed using separate tests), we were able to calculate the amount of time spent between the user program and DDP, i.e., the time spent in the

application-interface protocols. Since the amount of time spent in user mode is negligible, the remaining time represents that necessary to to perform the *write()* or *read()* system call, copy data to or from the kernel, traverse the kernel's socket layer or stream head and enter the protocol. Because users of both subsystems cannot avoid any of these parts, measuring its total impact is sufficient for our comparison. Our measurement of BSD/Sockets roughly corresponds with that found in [19]; we know of no similar measurements for TLI/Streams. Unfortunately, we were unable to measure this impact in the context of the *x*-Kernel because no such "standard" application-interface exists for its user-space implementation.

Our measurements demonstrate that, on the average, BSD and the *x*-Kernel outperform Streams for simple datagram-oriented protocols (such as DDP) and to a lesser degree for more complex, transaction-oriented protocols (such as ATP). BSD is similar to the *x*-Kernel in performance for ATP, while more efficient for DDP. The disparity (for Streams) is due, in part, to the fact that Streams protocols must spend considerable time and code unpacking and packing TLI messages. During the process of porting from Streams to BSD we noticed that several hundred lines of code could be attributed to this task. The disparity (for the *x*-Kernel) is primarily due (we believe) to data copying necessitated by the *x*-Kernel's message interface[12] and the overhead imposed by the underlying threads package (SunOS LWP). As the amount of protocol processing increases (in more complex protocols like ATP, for example) this disparity becomes less of an impact.

Part of the disparity of Streams can also be attributed to the usage of service queues which add context-switch overhead to protocol processing. Also contributing is overhead involved in allocating and deallocating Streams buffers when compared to BSD *mbufs* and the *x*-Kernel *BufferPool*. To get an idea of the performance impact of buffer manipulation, we measured the time necessary to allocate and free $10,000$ buffers in each subsystem. We chose a buffer size[13] roughly equivalent to an "average" size AppleTalk header or Streams TLI message header. We calculated the values in much the same manner as earlier tests, taking multiple samples and averaging over the range of values. We measure the time to both allocate and deallocate together rather than separating them because all subsystem must eventually free previously allocated buffers. Therefore, lumping the two measurements together still allows us to compare the overall impact of buffer manipulation. The results, depicted in Figure 4, are somewhat startling with a disparity of almost $8:1$ between the *x*-Kernel/BSD and Streams. Part of this disparity results from the fact that the *mbuf* structure contains room for a small amount of data (usually sufficient for a few protocol headers) while the *strbuf* structure does not. Allocating a *strbuf* entails the retrieval of a free *strbuf* and the allocation and attachment of a data buffer. *x*-Kernelbuffer data is preallocated and cached so little cost is incurred when allocating and deallocating.

We also conclude that a significant performance degradation can be attributed to application-interface protocol processing. Although this fact is nothing new, what surprised us was that the performance penalty when using TLI dwarfs that of Sockets (over $6:1$) and accounts for as much as 95% of the protocol processing for our Streams protocol graph. Obviously, given current operating system architectures, some portions of this delay, mainly copying to and from the kernel as well as context switching overhead, are unavoidable.

One could argue that this impact (along with buffer allocation and de-allocation overhead) is more a matter of implementation efficiency rather than architectural or subsystem limitations. To some degree,

[12]Messaging, in version 3.2 of the *x*-Kernel hides all knowledge of its implementation behind an abstract data type; while convenient for programming, performance can suffer because protocols must copy headers out of the message instead of accessing them directly. A future version of the *x*-Kernel is supposed to offer more efficient alternatives.

[13]For BSD, we allocated a single 128 byte *mbuf* structure; for Streams, we allocated a 64 byte message block; for the *x*-Kernel, we allocated a 64-byte BufferPool block.

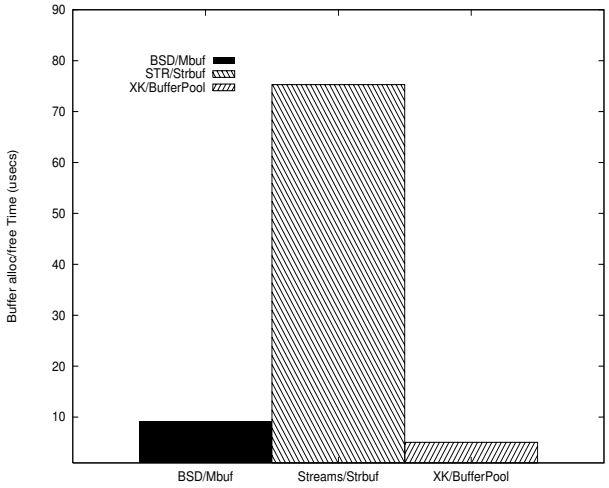

Figure 4: Buffer Alloc/Free Timings (usecs)

this argument is accurate. However, there are a few items worth mentioning that are indirectly related to the subsystem architecture rather than the level it has been tuned. Streams horizontal process architecture limits the ability to keep pointers to TLI message blocks because the protocol programmer does not know when lower layer protocols have finished their processing. Within the vertical process architecture of BSD and the *x*-Kernel, we can assume that resources are no longer in use when we return from a lower-layer invocation. This difference lets us keep copies of structures around rather than performing an expensive allocation/deallocation for each packet. Second, the basic structure of an *mbuf* (which includes enough space for several protocol headers and data so that an additional memory allocation can potentially be avoided) directly translates into better performance regardless of the amount of subsystem tuning that has occurred. For interface protocol processing, similar conclusions can be drawn. The fact that TLI overhead dwarfs that of Sockets indicates that some amount of tuning can be performed[14]. However, the very nature of TLI (involving message allocation, packing, and unpacking) suggests that its performance will never be on par with Sockets.

6 Adapter-Based Multi-Subsystem Architectures

Following the motivation developed so far in this paper, we now discuss an approach that would allow us to build protocol graphs spanning multiple subsystems. We first describe some general aspects of our proposed approach and then describe our experience with it in the context of the case study.

[14]Indeed, correspondence with engineers from Sun indicate that TLI in SunOS has not been extensively tuned.

6.1 General Approach

We assume that the relevant subsystems co-exist in the same operating system kernel (much like BSD and Streams co-exist within many Unix kernels while the *x*-Kernel can run in user-space). Since supporting easy portability is of concern to us we adopt an approach that does not require changing protocol implementations designed to work within a single-subsystem environment. Rather we introduce a new protocol module, *the subsystem adapter*, which allows protocols to interconnect across subsystem boundaries. Adapter protocols are situated at or on the boundary between subsystems for which they provide adaptation services. For example, Figure 5a depicts an adapter protocol (labeled 'A') situated between two protocols in two distinct subsystems. A more detailed look at the adapter protocol is shown in Figure 5b.

An adapter protocol design and implementation must satisfy one or both of the following properties:

1. If the multi-subsystem configuration is being used to avoid having to port protocols among subsystems, then either "ready-made" adapter protocols must be available or the implementation of a new adapter protocol must be considerably simpler than the process of protocol porting.
2. If the multi-subsystem configuration is being used because it is desired for performance or other reasons[15], then the adapter protocol must not represent a significant performance overhead and/or must not affect the desired subsystem feature(s).

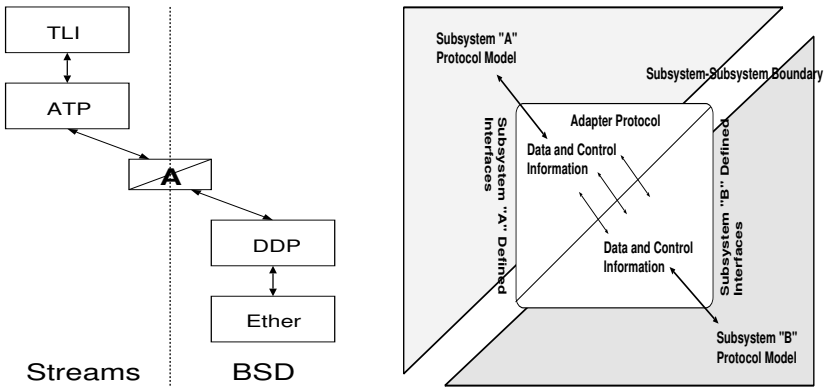

Figure 5: Adapter Protocol Design

Adapter protocols share some features with application interface protocols (such as TLI and Sockets). Both types of protocol provide an interface from outside a protocol subsystem to within. While application interface protocols are generally relegated to the "top" of protocol graphs and provide an interface between the subsystem and the greater system in which it is embedded, adapter protocols exist

[15] such as the configurability features of Streams.

at any point in the protocol graph where subsystem boundaries are crossed. Adapter protocols are also similar to the virtual protocols defined in the *x*-Kernel research [18]. Both work transparently within a protocol graph. They do not produce or consume protocol headers nor communicate with remote peers. Like virtual protocols, an adapter need not have a remote peer. Subsystem adapters are also similar in nature to protocol adaptors defined by Calvert [5] except that subsystem adapters do not alter or convert protocol headers, but instead provide subsystem adaptation services.

Adapter protocols reside in the subsystems between which they operate. The manner in which an adapter protocol is integrated into each subsystem depends on the subsystem in question. In general, adapter protocols operate by presenting themselves as (or proxying for) the protocols (and subsystems) for which they provide adaptation services. For example, in Figure 5a, the adapter protocol situated between ATP_{STR} and DDP_{BSD} presents itself as DDP to ATP and the Streams subsystem while simultaneously presenting itself as ATP to DDP and the BSD subsystem. Separate adapter protocols are necessary for each pair of subsystems. For example, the adapter protocol used between Streams and BSD would not be applicable to a protocol graph spanning Streams and *x*-Kernel.

The main purpose of an adapter protocol is to understand the protocol model imposed by each subsystem and to provide adaptation or conversion services between them. In particular, they must be capable of supporting all relevant control and data interfaces, process architecture, buffer conventions, and support services for each subsystem. In many cases, protocols and subsystems may be sufficiently different that certain translations may not be possible. For example, a protocol in one subsystem may not support certain parameter negotiation required by a protocol in another subsystem. If both protocols are sufficiently robust, interconnection may still be possible. In others cases, interconnection may be impossible if that parameter negotiation is important for the fundamental operation of one protocol. In this instance, the adapter protocol is faced with a fundamental choice: it can return an error when a particular translation is not possible, or it may return success despite the fact that the translation did not occur and is not possible. The choice depends on the nature of the translation, its importance, and its side-effects.

Because each adapter protocol performs similar operations and shares a common design, their implementation presents a good candidate for the object-oriented development techniques of inheritance and specialization. That is, once one implements a generic adapter protocol for a given pair of subsystems, only a small amount of specialization is necessary for any two protocols in those subsystems. For example, once our adapter protocol in Figure 5(a) is coded, only a small amount of code is necessary for it to adapt between ATP_{STR} and DDP_{BSD}. This specialization is necessary because each protocol (although still conforming to the BSD and Streams subsystems) approachs argument and option passing differently. In future work we plan to take a closer look using inheritance and specialization as an implementation methodology.

6.2 A BSD-Streams Adapter Protocol

We implemented a BSD-Streams adapter protocol[16] allowing for the interconnection of AppleTalk's ATP and DDP across subsystem boundaries. We coded our adapter protocol in such a way that it could

[16]During the course of our research we implemented three adapters: BSD-Streams, *x*-Kernel-Streams, and *x*-Kernel-BSD. We focus on one adapter, BSD-Streams, because it is representative of all our adapter protocols.

be used when providing both $ATP_{STR} \Longleftrightarrow DDP_{BSD}$ and $ATP_{BSD} \Longleftrightarrow DDP_{STR}$ services. In all cases, the use of our BSD-Streams (and our *x*-Kernel-Streams and *x*-Kernel-BSD as well) adapter protocol satisfied our first requirement: namely, no existing protocol implementation should require modification in order to build multi-subsystem protocol graphs. We next highlight the operation of our BSD-Streams adapter protocol. We examine the effect of adapters on protocol performance in the next subsection. First, we discuss some aspects of one of our adapter implementations.

Adapting Disparate Protocol Models The first job of our BSD-Streams adapter protocol is to translate between the protocol models imposed by Streams and BSD. Because BSD defines several different protocol models, our adapter protocol bases its decision (as to which protocol model to convert to) on the protocols it interconnects. For example, since we are providing adaptation services between two protocols (ATP and DDP) which are not network-interface protocols nor application interface layer protocols, the choice was fairly obvious. Had we decided to implement an adapter protocol between DDP_{STR} and a BSD network-interface protocol, our adapter protocol would have had to convert between a different BSD protocol model.

Next, our BSD-Streams adapter protocol had to translate the protocol interfaces of each subsystem. For example, BSD's data input and output functions roughly corresponded to Streams's read and write functions. Data entering from the Streams-subsystem enters a Streams compatible read or write interface; likewise, data entering from the BSD subsystem enters a BSD compatible input or output function. Control flow, on the other hand, is not so easy to translate. Recall that control information, in BSD, flowed through differing interfaces based on its origin; in Streams, all control information flowed in-band. When converting control information originating in Streams and destined for BSD, our adapter protocol had to make a decision as to which BSD interface to use. All control information flowing in the other direction (from BSD to Streams) utilizes the same interface.

The Streams subsystem sometimes invokes inter-module flow control (via subsystem-specific control messages) because it utilizes an asynchronous, message-passing style of protocol invocation. This feature is necessary so that a Streams protocol can throttle higher or lower layer protocols. Inter-module flow control occurs in two forms: per-module (via high/low water marks on queues) and per-protocol-graph (via *flush* control messages). Because BSD does not provide (nor require) equivalent functionality, our adapter protocol does not attempt to translate inter-module flow control when data arrives from Streams towards BSD. Inter-module flow control is supported in one direction only – from BSD to Streams. As data flows toward the Streams subgraph, the adapter protocol examines the incoming queue of its adjacent Streams protocol; if it is full, the adapter protocol will queue the data.

In order for our BSD-Streams adapter protocol to properly function, it must reside in both subsystems. Consequently, an entry is placed in the BSD protocol switch table as well as the Streams configuration table. For Streams, it registers its Streams compatible interfaces and data structures (e.g., *open*, *close*, and *streamtab* structure); for BSD, it registers its BSD compatible interfaces (e.g., *usrreq*, *input*, and *protosw* structure). When the BSD subsystem is initialized at kernel boot time, our adapter protocol is initialized along with any other BSD protocol. Likewise, when the Streams protocol graph is constructed, our adapter protocol is dynamically linked in.

Adapting Disparate Process Architectures If the two subsystems, between which an adapter protocol operates, support a common process architecture then that adapter protocol's operation is somewhat simplified. Unfortunately, BSD and Streams differ slightly in their process architectures. Recall that BSD utilized a vertical process architecture while Streams utilized a combination of vertical and horizontal. When packets arrive from Streams, our adapter protocol unbundles them and invokes the appropriate BSD protocol function. When packets arrive from BSD and are headed for Streams, our adapter protocol creates a TLI message, and passes it on to the appropriate Streams protocol; an interesting side-effect occurs. When BSD protocol function invocations finish, an error value (eventually becoming the UNIX *errno* variable) is returned. This error message (failure or success) eventually cascades back to the user. In Streams no such immediate error feedback is returned. Therefore, when a return value does enter the adapter protocol, it progresses no further. In affect, when data leaves the BSD subsystem and enters Streams our adapter protocol returns success (setting the BSD return value to 0) even though it does not know the eventual fate of that packet. Alternatively, when a Streams error message enters the adapter protocol, it also progresses no further.

Lastly, we mention functionality that our adapter protocol does *not* need to perform. Recall that subsystems often provide support services (e.g., routing table and multiplexing support). Unlike the task of porting protocols, adapter protocols need not provide any such translation between differing subsystem support services because all protocols in multi-subsystem protocol graphs continue to operate within the subsystem for which they were originally coded. Because original protocols remain unmodified, the use of adapter protocols should have no effect on established operational properties like correctness, conformance, or interoperability.

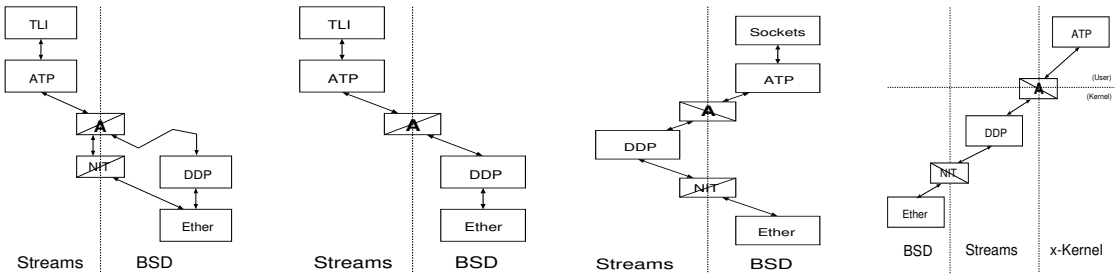

Figure 6: Multi-Subsystem Protocol Graphs

Our adapter protocols allowed us to construct the multi-subsystem configurations of the AppleTalk protocol architecture shown in Figure 6. The BSDSTR configuration maintains DDP in the Streams subsystem with the rest of the protocol graph in the BSD subsystem while the XKSTR configuration maintains ATP in the *x*-Kernel with the rest of the protocol graph residing in Streams. The resulting protocol graphs were fully tested for inter-operability with our previous single-subsystem implementations (BSDATALK, STRATALK, and XKATALK depicted in Figure 2), with other multi-subsystem protocols, and with the native Macintosh protocol implementations.

We constructed two variations of the STRBSD (STRBSD and STRBSD') protocol graphs. Their difference involves the manner in which Streams protocols access network interface protocols in SunOS. Since Streams protocols cannot directly access network interface protocols (e.g., ethernet) in SunOS (because the latter are coded within the BSD subsystem), they must utilize the network interface tap (NIT) pseudo-device to do so[17]. Because our adapter protocol appears as DDP to Streams, and because of our goal not to modify existing protocol software, DDP is normally linked on top of the NIT pseudo-device when that protocol graph is first constructed. However, in a STRBSD multi-subsystem protocol graph, the services of the NIT device are not necessary because DDP_{BSD} can access the network interface natively since it resides within the BSD subsystem. For completeness though, we measured STRBSD with and without NIT in the protocol graph. Our measurements showed little, if any, difference between the two variations.

6.3 Multi-subsystem Protocol Graph Performance

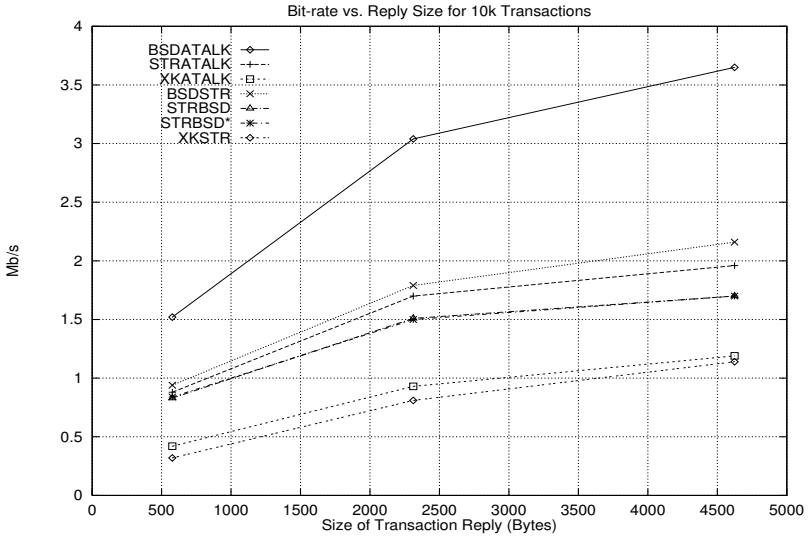

Figure 7: Protocol Graph Performance in Loopback Mode

We conducted our performance measurements on the various protocol graphs we were able to construct (see Figures 6 and 2) using a simple client and server. 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. Separate tests were conducted, each consisting of the client invoking $10,000$ transactions. Each transaction request consisted of a small, 45 byte packet while replies ranged from 578 (1 packet) to 4624 (8 packets) bytes. As before these test were conducted in loopback mode. The results are depicted in Figure 7.

Our results clearly demonstrate that different protocol graphs do indeed produce different results. Not surprisingly, the native BSD protocol graphed performed best. Somewhat surprising was the fact

[17]In affect, the NIT module can be thought of as a primitive adapter protocol because it performs limited translation services between the Streams and BSD subsystems.

that the native and hybrid *x*-Kernel protocol graphs performed worst; the Streams protocol graphs fared slightly better. We attribute the poor performance of the *x*-Kernel protocol graphs to the underlying threads package (SunOS LWP) rather than the *x*-Kernel. Also surprising was the fact that the BSDSTR graph out-performed the STRBSD graph. This difference can be attributed to the poorer performance of TLI (also affecting the native Streams protocol graph) when compared to Sockets. The STRBSD graph utilizes TLI (since a Streams protocol is "on top") whereas the BSDSTR graph utilizes Sockets (since a BSD protocol is "on top"). Recall from Figure 3 the substantial difference in the performance of Sockets and TLI.

Further, our results demonstrate that some multi-subsystem protocol graphs offer the best of both worlds. The BSDSTR protocol graph successfully combines the performance of the BSD subsystem with the dynamic protocol graph configurability of Streams. Using a BSDSTR approach, one could build protocol graphs avoiding the poorer performance of TLI yet still benefitting from the ability to change the Streams portion of the protocol graph at run-time. The relative performance of the three multi-subsystem protocol graphs (STRBSD, BSDSTR, and XKSTR) does present interesting questions, however. Why do some multi-subsystem protocol graphs perform better than others? What, if any, performance overhead is introduced by adapter protocols? We explore these questions below.

Adapter Protocol Performance In order to assess the performance impact introduced by adapter protocol use, we conducted tests designed to measure the protocol processing time required to provide adaptation services between the subsystems. As before, we placed code at all relevant entry and exit points and recorded the arrival and departure times for each packet. We then utilized our client and server from previous tests and performed $10,000$ transactions. Figure 8 presents our results. The poor performance of the *x*-Kernel-Streams (vs. the BSD-Streams and Streams-BSD adapters) is a result of several things. First, the XKSTR protocol graph straddles the user/kernel boundary so context-switching costs creep into the performance of the *x*-Kernel-Streams adapter protocol. The BSDSTR and STRBSD protocol graphs and associated adapter protocols reside entirely within the kernel so this cost does not creep into adapter protocol performance measurements. Second, the performance of the SunOS LWP threads library creeps into the *x*-Kernel-Streams adapter protocol measurements as well.

We draw several conclusions about the performance of adapter protocols.

1. From Figure 8, one can clearly see that the cost of going from BSD to Streams exceeds its opposite. We attribute this disparity to the fact that transforming from Streams to BSD requires no data copy because BSD's buffer facility supports the importation of data without copying while Streams does not. Transformations to and from the *x*-Kernel also entails data copying due to the fact that the *x*-Kernel resides in user-space on SunOS systems. However, the *x*-Kernel buffer facility does support the importation of data without copying. Therefore, if the *x*-Kernel resided in the space protection domain as Streams and BSD, data-copying costs would not affect its performance.

2. Transformation costs are dependent on the individual protocols in question and their position in the protocol graph. For example, at first we hypothesized that all BSD to Streams transformation should be equivalent; all Streams to BSD transformations likewise. Our results do show that both Streams to BSD transformations performed similarly (145 and 128 usecs). However,

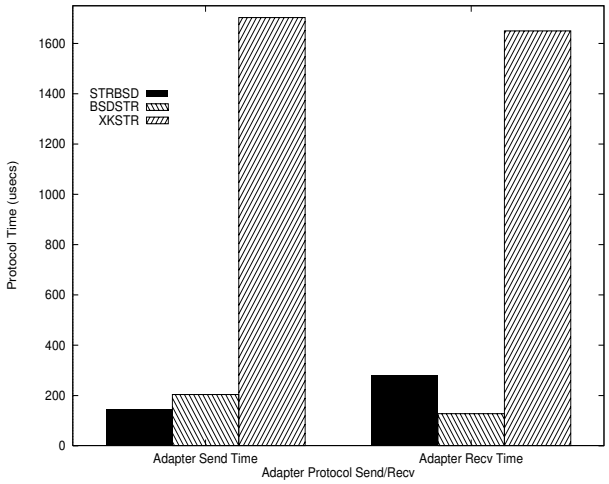

Figure 8: Adapter Protocol Timings (usecs)

BSD-Streams transformations did not. We attribute the performance difference in BSD-Streams transformations to the overhead involved in copying data from BSD *mbufs* to Streams *strbufs*. In the BSDSTR protocol graph, the BSD-Streams translation copies less data than a similar translation in STRBSD graph due to its "higher" (less data and headers have been added to the packet) position in the protocol graph. Likewise, the position at which adaptation services occur in relation to fragmentation/reassembly services may also affect performance. In order to reduce adaptation overhead, one would want to perform adaptation services after reassembly on the receiving side and before fragmentation on the sending side.

3. We conclude that the overhead introduced by adapter protocol usage is fairly consistent and increases linearly with the number and size of packets. The overhead is not fixed, however, due to the fact that data copying is necessary for all BSD-Streams transformations.

7 Conclusion

Protocol subsystems can have a profound effect on the structure and capabilities of protocol implementations. Differences among subsystems can also influence the protocol porting process and, as a byproduct, can influence the ease of adoption of new protocols. In this paper we have investigated issues arising from differences in subsystems and their effects on protocol portability and performance. We use the resultant understanding to motivate the desirability, in certain situations, of protocol graphs spanning multiple subsystems. We propose a possible adapter-based approach to constructing such protocol graphs and report on our experience of constructing adapter-based implementations within the context of our case study. Our experiments indicate that the approach has promise, in particular it can increase flexibility and protocol code reuse with a modest performance cost.

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] AT&T. *UNIX System V/386 RELEASE 4 STREAMS Programmer's Reference Manual*. Prentice-Hall Inc., 1991.

[3] Joshua Auerbach. A protocol conversion software toolkit. In *ACM SIGCOMM-1989 Symposium*, pages 259–270, September 1989.

[4] S. Bradner and A. Mankin. The recommendation for the IP next generation protocol. Anonymous FTP, January 1995. RFC 1752.

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

[12] 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.

[13] S.S. Lam. Protocol conversion. *IEEE Transactions on Software Engineering*, 14(3):353–62, March 1988.

[14] 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.

[15] Allen B. Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebsting, and John H. Hartman. Scout: A communications-oriented operating system. Technical Report TR94-20, Department of Computer Science, University of Arizona, June 1994.

[16] Bodhisattwa Mukherjee, Greg Eisenhauer, and Kaushik Ghosh. A machine independent interface for lightweight threads. *Operating Systems Review*, 28(1):33–47, January 1994.

[17] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Performance issues in parallelized network protocols. In *Proceedings of the First Symposium on Operating System Design and Implementation*. Department of Computer Science, University of Massachusetts, 1994.

[18] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10:110–143, May 1992.

[19] 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.

[20] Craig Partridge and Stephen Pink. A faster UDP. *IEEE/ACM Transactions on Networking*, 1(4):429–440, August 1993.

[21] Patric Peters, Roy Dcruz, Chiun-Teh Sung, Christine Wang, and Branislav Meandzija. On generalizations in networking software to encourage code portability. In *Proceedings of IEEE INFOCOM*, pages 261–267, 1989.

[22] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, October 1984.

[23] 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.

[24] Douglas C. Schmidt and Tatsuya Suda. Measuring the performance of parallel message-based process architectures. In *Proceedings of IEEE INFOCOM*, 1995.

[25] Gursharan S. Sidhu, Richard F. Andrews, and Alan B. Oppenheimer. *Inside AppleTalk*. Addison-Wesley, 1st edition, 1989.

[26] 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. Also published in SIGCOMM '93.